(NASA-TM-X-74029) ANOPP PROGRAMMER'S REFERENCE MANUAL FOR THE EXECUTIVE SYSTEM (NASA) 380 p HC A17/MF A01 CSCL 09B

N77-22846

Unclas
G3/61 24946

NASA TECHNICAL MEMORANDUM

NASA TM X- 74029

NASA TM X- 74029

ANOPP Programmers' Reference Manual for the Executive System

by

Ronnie E. Gillian, Christine G. Brown, Robert W. Bartlett,
and Patricia H. Baucom

APRIL 1977

This informal documentation medium is used to provide accelerated or special release of technical information to selected users. The contents may not meet NASA formal editing and publication standards, may be revised, or may be incorporated in another publication.

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

1. Report No. TM X-74029	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle ANOPP Programmers' Reference Manual for the Executive System		5. Report Date April 1977 6. Performing Organization Code
7. Author(s) Ronnie E. Gillian, Christine G. Brown, *Robert W. Bartlett, and*Patricia H. Baucom		8. Performino Oroanization Report No.
9. Performing Organization Name and Address NASA Langley Research Center Hampton, Virginia 23665		10. Work Unit No. 505-03-21 11. Contract or Grant No.
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D. C. 20546		13. Type of Report and Period Covered Technical Memorandum 14. Sponsoring Agency Code

15. Supplementary Notes

*Mr. Bartlett and Ms. Baucom are members of the Control Data Corporation.

16. Abstract

The ANOPP Programmers' Reference Manual for the Executive System embodies the documentation for ANOPP as of release level 01/00/00. The manual is designed for users who have need for understanding the internal design and logical concepts of the ANOPP Executive System software. Emphasis is placed on providing sufficient information to the programmer to modify the system for enhancements or error correction.

The ANOPP Executive System includes software related to operating system interface, executive control, and data base management for the Aircraft Noise Prediction Program. It is written in Fortran IV for use on CDC Cyber series of computers.

17. Key Words (Suggested by Author(s)) ANOPP	18. Distribution Statement Unclassified - Unlimited

19. Security Classif. (of this report) Unclassified	20. Security Classif. (of this page) Unclassified	21. No. of Pages 380	22. Price* $10.75

* For sale by the National Technical Information Service, Springfield, Virginia 22161

PREFACE

The ANOPP Programmers Reference Manual For Executive System embodies the documentation in its entirity for ANOPP as of release level 01/00/00. Additional manuals are anticipated in order to satisfy the various needs of the ANOPP user community. These anticipated manuals include the following:

Theoretical Manual

User's Manual

Demonstration Problem Manual

Functional Module Writer's Guide

The Programmers Reference Manual is designed for usage by those who have need for understanding internal design and logical concepts of the ANOPP Executive System. Emphasis has been placed on providing sufficient information to the programmer in order to modify the system in pursuit of either enhancements or error correction.

ANOPP PROGRAMMERS REFERENCE MANUAL FOR

EXECUTIVE SYSTEM

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

PAGE STATUS LOG

Page No.	Most Recent Date Changed	Page No.	Most Recent Date Changed	Page No.	Most Recent Date Changed
i		2.3-1		3.3-20	
ii		2.3-2		3.3-21	
iii		2.3-3		3.3-22	
iv		2.3-4		3.3-23	
v		2.3-5		3.3-24	
vi		2.3-6		3.3-25	
vii		2.3-7		3.3-26	
viii		2.3-8		3.3-27	
ix		2.3-9		3.3-28	
x		2.3-10		3.3-29	
xi		2.3-11		3.3-30	
xii		2.3-12		3.3-31	
xiii		2.3-13		3.3-32	
xiv		2.3-14		3.3-33	
xv		2.3-15		3.4-1	
xvi		2.3-16		3.4-2	
1.1-1		2.4-1		3.4-3	
1.2-1		2.4-2		3.4-4	
1.3-1		2.4-3		3.4-5	
1.3-2		2.5-1		3.4-6	
1.4-1		2.5-2		3.4-7	
1.4-2		2.5-3		3.4-8	
1.5-1		2.5-4		3.4-9	
1.6-1		2.5-5		3.4-10	
1.6-2		2.5-6		3.4-11	
1.6-3		2.5-7		3.4-12	
1.7-1		2.5-8		3.4-13	
1.7-2		2.5-9		3.4-14	
1.8-1		2.5-10		3.4-15	
1.8-2		3.1-1		3.4-16	
1.8-3		3.2-1		3.4-17	
1.8-4		3.2-2		3.4-18	
1.8-5		3.2-3		3.4-19	
1.8-6		3.2-4		3.4-20	
1.9-1		3.2-5		3.4-21	
1.10-1		3.2-6		3.4-22	
2.1-1		3.3-1		3.4-23	
2.2-1		3.3-2		3.4-24	
2.2-2		3.3-3		3.4-25	
2.2-3		3.3-4		3.4-26	
2.2-4		3.3-5		3.4-27	
2.2-5		3.3-6		3.4-28	
2.2-6		3.3-7		3.5-1	
2.2-7		3.3-8		3.5-2	
2.2-8		3.3-9		3.5-3	
2.2-9		3.3-10		3.5-4	
2.2-10		3.3-11		3.5-5	
2.2-11		3.3-12		3.5-6	
2.2-12		3.3-13		3.5-7	
2.2-13		3.3-14		3.5-8	
2.2-14		3.3-15		3.5-9	
2.2-15		3.3-16		3.5-10	
2.2-16		3.3-17		3.5-11	
2.2-17		3.3-18		3.5-12	
2.2-18		3.3-19		3.5-13	

PAGE STATUS LOG

Page No.	Most Recent Date Changed	Page No.	Most Recent Date Changed	Page No.	Most Recent Date Changed
3.5-14		3.5-69		3.6-30	
3.5-15		3.5-70		3.6-31	
3.5-16		3.5-71		3.6-32	
3.5-17		3.5-72		3.6-33	
3.5-18		3.5-73		3.6-34	
3.5-19		3.5-74		3.6-35	
3.5-20		3.5-75		3.6-36	
3.5-21		3.5-76		3.6-37	
3.5-22		3.5-77		3.6-38	
3.5-23		3.5-78		3.6-39	
3.5-24		3.5-79		3.6-40	
3.5-25		3.5-80		3.6-41	
3.5-26		3.5-81		3.6-42	
3.5-27		3.5-82		3.6-43	
3.5-28		3.5-83		3.6-44	
3.5-29		3.5-84		3.6-45	
3.5-30		3.5-85		3.6-46	
3.5-31		3.5-86		3.6-47	
3.5-32		3.5-87		3.6-48	
3.5-33		3.5-88		3.6-49	
3.5-34		3.5-89		3.6-50	
3.5-35		3.5-90		3.6-51	
3.5-36		3.5-91		3.6-52	
3.5-37		3.5-92		3.6-53	
3.5-38		3.5-93		3.6-54	
3.5-39		3.5-94		3.6-55	
3.5-40		3.6-1		3.6-56	
3.5-41		3.6-2		3.6-57	
3.5-42		3.6-3		3.6-58	
3.5-43		3.6-4		3.6-59	
3.5-44		3.6-5		3.6-60	
3.5-45		3.6-6		3.7-1	
3.5-46		3.6-7		3.7-2	
3.5-47		3.6-8		3.7-3	
3.5-48		3.6-9		3.7-4	
3.5-49		3.6-10		3.7-5	
3.5-50		3.6-11		3.7-6	
3.5-51		3.6-12		3.7-7	
3.5-52		3.6-13		3.7-8	
3.5-53		3.6-14		3.7-9	
3.5-54		3.6-15		3.7-10	
3.5-55		3.6-16		3.7-11	
3.5-56		3.6-17		3.7-12	
3.5-57		3.6-18		3.7-13	
3.5-58		3.6-19		3.7-14	
3.5-59		3.6-20		3.7-15	
3.5-60		3.6-21		3.7-16	
3.5-61		3.6-22		3.7-17	
3.5-62		3.6-23		3.7-18	
3.5-63		3.6-24		3.7-19	
3.5-64		3.6-25		3.7-20	
3.5-65		3.6-26		3.7-21	
3.5-66		3.6-27		3.7-22	
3.5-67		3.6-28		3.7-23	
3.5-68		3.6-29		3.7-24	

PAGE STATUS LOG

Page No.	Most Recent Date Changed	Page No.	Most Recent Date Changed	Page No.	Most Recent Date Changed
3.7-25		3.9-18		B.1-13	
3.7-26		3.9-19		B.1-14	
3.7-27		3.9-20		B.1-15	
3.7-28		3.9-21		C.1-1	
3.7-29		3.9-22		C.1-2	
3.7-30		3.9-23		C.1-2	
3.7-31		3.9-24		C.1-3	
3.7-32		3.9-25		C.1-4	
3.8-1		3.9-26		C.1-5	
3.8-2		3.9-27		C.1-6	
3.8-3		3.9-28		C.1-7	
3.8-4		3.9-29		C.1-8	
3.8-5		3.9-30		C.1-9	
3.8-6		3.9-31		C.1-10	
3.8-7		3.9-32		C.1-11	
3.8-8		4.1-1		C.1-12	
3.8-9		4.2-1		C.1-13	
3.8-10		4.2-2		C.1-14	
3.8-11		4.2-3		C.1-15	
3.8-12		4.2-4		C.1-16	
3.8-13		4.2-5		D-1	
3.8-14		4.2-6			
3.8-15		4.2-7			
3.8-16		4.2-8			
3.8-17		4.2-9			
3.8-18		4.2-10			
3.8-19		4.2-11			
3.8-20		4.2-12			
3.8-21		4.2-13			
3.8-22		4.2-14			
3.8-23		4.2-15			
3.8-24		4.2-16			
3.8-25		4.2-17			
3.8-26		4.2-18			
3.8-27		4.2-19			
3.8-28		A-1			
3.8-29		A-2			
3.8-30		A-3			
3.9-1		A-4			
3.9-2		A-5			
3.9-3		A-6			
3.9-4		A-7			
3.9-5		A-8			
3.9-6		B.1-1			
3.9-7		B.1-2			
3.9-8		B.1-3			
3.9-9		B.1-4			
3.9-10		B.1-5			
3.9-11		B.1-6			
3.9-12		B.1-7			
3.9-13		B.1-8			
3.9-14		B.1-9			
3.9-15		B.1-10			
3.9-16		B.1-11			
3.9-17		B.1-12			

INTRODUCTION

1.1 PROGRAM OVERVIEW

The Aircraft Noise Prediction Program (ANOPP) was developed to be a repository for current and future approaches to computerized study of aircraft noise. Today's methods are highly empirical; tomorrow's will be analytical. To the developer of new technology and prediction methods, ANOPP is where new algorithms and code can be deposited as a new or replacement part of an integrated system. To the planner or user, ANOPP is the source of current information and state-of-the-art prediction methods at selected levels of complexity for a suitably described model.

With these objectives in mind, the fundamental design requirements of the system were defined as:

1. flexibility for the addition, replacement, or removal of prediction methods
2. user control for selective and effective use of the various methodologies

The design requirements are met by separating executive functions from noise prediction functions. Thus, all of the noise prediction applications technology is contained in functional modules.

The executive system provides for program initialization and interface with the host computer operating system. It provides for user control of execution via a control statement language processor. It provides storage management and data management for the functional modules. It provides error handling and exit procedures to the host operating system.

The remainder of this chapter will outline the characteristics and motivations for several significant parts of the executive system. The interfaces and interactions between the executive and functional modules will be shown. The rest of this Programmer's Manual will provide more individual detail of the entire executive system.

1.2 FUNCTIONAL MODULE CONCEPT

A functional module is a logically independent group of subprograms or modules which performs noise prediction functions. The size varies with the number and type of modules required for the functions to be performed.

A functional module is called into execution by the ANOPP executive system upon user request via a control statement. At the time of request the specified functional module is loaded into core and execution is begun. Upon completion the functional module returns execution control to the ANOPP executive system. An executive module is brought into the core space previously occupied by the now completed functional module to process the remaining control statements supplied by the user. Upon encountering a subsequent functional module request, the process is repeated.

Thus a functional module is core resident if and only if it is being executed at user request. It is transient and shares the same core space with other functional modules as well as executive modules.

General characteristics of functions include:

1. Functional modules are independent.
2. Functional modules do not call one another directly.
3. Functional modules are called from and return control to the executive manager.
4. Functional modules request and release storage through a dynamic storage manager.
5. Functional modules input and output data through a data base manager.

1.3 CONTROL STATEMENT CONCEPT

Just as the loading and execution of ANOPP is controlled by a set of job control cards monitored by the host computer operating system, so is the sequence of module executions within ANOPP controlled by a set of executive control statements monitored by the ANOPP executive management system. The control statements interpreted by the ANOPP executive manager provide:

1. execution sequence control with branching
2. exchange of parameters among the user, the executive manager, and the functional modules
3. update capabilities for the ANOPP data base
4. the ability to load/unload various parts of the ANOPP data base.

The format for an ANOPP control statement is:

```
label   control statement name    operand(s)   $    optional comment(s)
```

The structure and operand(s) appropriate to specific control statements can be found in Section 3.5.2 of this manual. Several general characteristics are of interest here. The label field provides tag addresses for looping and branching. The operands provide parameters for control of conditional branching and exchange of information among the user, the executive manager, and the functional modules. Control statements are terminated with a dollar sign ($); otherwise, they are assumed to be continued on the next card image. A limited number of continuation cards are permitted. Optional comments may follow the terminal character.

A set of control statements in card image format is taken from the primary input stream, edited and stored in edited format on the data base. The statements are then executed one at a time from this edited set. A collection of control statements in unedited or card image form may be saved in the ANOPP data base and subsequently retrieved by a CALL Control Statement in a later run. At the time of first execution of a CALL control statement, a specified pre-stored set of card image control statements is retrieved, edited and executed from beginning to end. Upon completion of the called set, execution then continues with the control statement following the CALL. The control statements in edited form are saved for subsequent execution if required. The called set may itself contain further CALL statements in a cascading series of expansions, but the user must be careful to avoid an infinitely recursive CALL loop. While not currently implemented, the capability to interactively enter and interpret single control statements in a card image set is not precluded in the present design of the executive system.

1.4 DYNAMIC CORE CONCEPT

Dynamic core is that portion of machine memory available within the program's region or field length that is not occupied by permanent or transient routines. This area is managed by a part of the ANOPP executive known as the dynamic storage manager. The total size of this dynamic core area varies directly with the field length available during an individual ANOPP execution. The capability provided by the dynamic storage manager is similiar to variable dimension arrays in FORTRAN and permits module writers to vary the size of data areas and to adjust their solution algorithms depending on the amount of core available.

The total dynamic core area is divided into two parts: a Global Dynamic Storage area and a Local Dynamic Storage area. Global dynamic storage is of fixed length during a single ANOPP execution and resides at the end of the program's region or field length. Local dynamic storage is of varying length and is bounded by the longest current transient routine on one side and by the start of global dynamic storage on the other side (see Figure 1). The dynamic storage manager allocates and de-allócates various size blocks up to the limits of the reserved space available for both local and global storage. Individual blocks of core within dynamic storage are defined by their starting location and length. The starting location is defined as an index relative to the variable IX in a fixed common block /XANØPP/ that resides near the beginning of the program's region. The global storage area is available throughout a single ANOPP execution and is generally used for executive tables and control blocks required by the executive system. The local storage area is reserved and released during individual executive or functional module executions and is generally available as scratch space during the execution of these transient routines. While functional modules may use available space in global storage during their time of execution, they can not use global storage blocks as a means or mechanism for transmitting information or data to other functional modules directly.

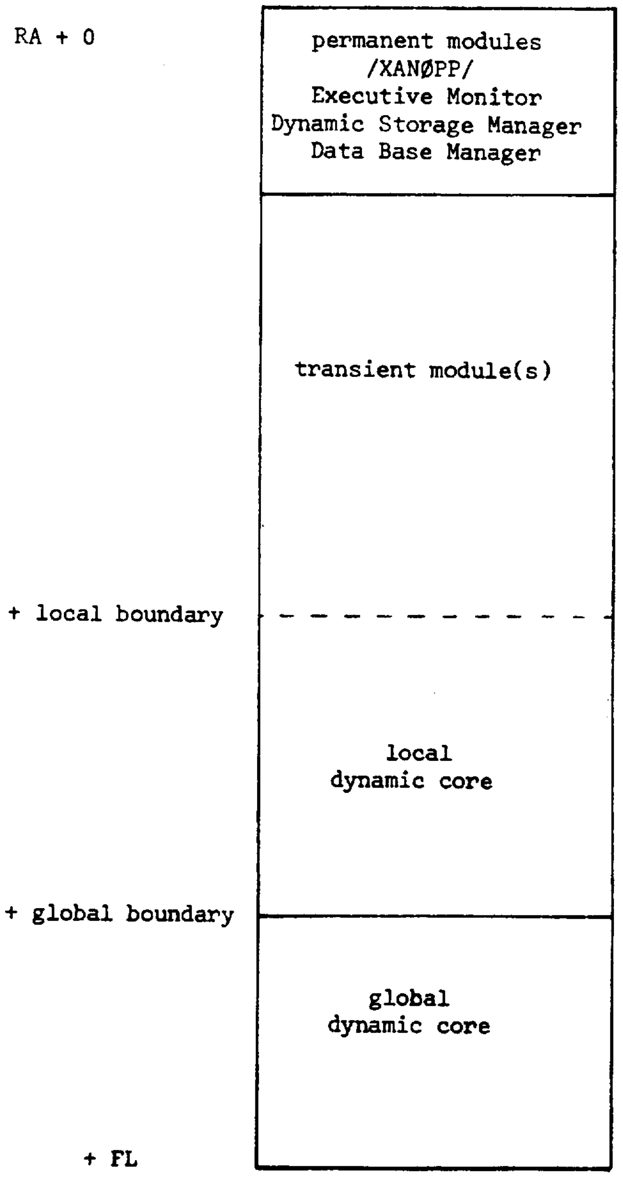


Figure 1. Layout of Core Storage.

1.5 ANOPP INPUT/OUTPUT

The Input and Output files from the host computer operating system are readily available to the routines of the ANOPP executive system. They are less easily accessible by the functional modules.

Functional modules receive and transmit their primary input/output via the ANOPP data base and the member manager and table manager facilities. There is no provision for reading directly from the host system input file. Information may be written directly to the host system output file; however, this should be done in conjunction with the executive system paging routines (XPLINE, XPAGE, etc.) to insure accurate line counts and page headings. Functional modules can also exchange limited numbers of parameters via the PARAM control statements, the executive parameter functions (XGETP, XPUTP, XASKP), and the user parameter table.

Since functional modules need information from the ANOPP data base to operate, the capability to place information in the data base must be provided. Thus, three control statements, DATA, TABLE, and UPDATE, provide means for reading cards from the primary input stream and transferring the information into the ANOPP data base in various selected formats.

1.6 EXECUTIVE MANAGEMENT

The Executive Management System consists of the main executive monitor which controls the sequence of operation of the execution phases and the submonitors which control the operations within each phase. The first program executed is the executive monitor (XM) which immediately calls the XBS module to perform ANOPP system initialization.

The executive bootstrap module (XBS) first checks to see that the required initial ANOPP control statements exist and are positioned correctly. XBS then initializes the global portion of dynamic storage via the dynamic storage manager, and executive and data base management tables and directories are allocated and initialized in dynamic storage. XBS returns to XM which calls the XRT module to edit the set of ANOPP control statements in the primary input stream and write the edited set on the data base to be executed later by the executive control statement processor module (XCSP).

The executive module XRT edits the set of control statements from the primary input stream in one pass and writes them to the ANOPP data base in an edited control statement format that is structured for input to the executive control statement processing phase. In the edit phase, the syntax of each control statement is checked and labels are matched with their corresponding branching statements. The edit phase does not "execute" the control statements, but simply transforms them from card image to "executable" format. Then, the later processing modules can act more efficiently on the executable form that is well structured syntactically and contains label tables for efficient branching capability. Several control statements have optional input following them. These control statements are DATA, TABLE, and UPDATE; and their input is terminated by an END* control statement before the next regular control statement. In these cases, the XRT module puts such input data on the data base and provides linking information so that this data can be retrieved during execution processing. The edit phase is complete and XRT returns to XM when an ENDCS card is found in the input stream. XM next calls the executive control statement processor module (XCSP) to execute the edited control statements.

The control statement processing module retrieves from the data base the edited form of the first control statement in the primary input stream and calls upon an appropriate executive module to process it. Upon process completion, control returns to the XCSP module which continues with the iterative pattern of read/process until the pattern is terminated by the ENDCS statement. The ENDCS indicates the set of control statements supplied by the user in the primary input stream has been completely executed and ANOPP should be terminated. The module which is called by XCSP to process the ENDCS thus performs normal termination procedures for ANOPP and does not return to XCSP.

XCSP may be interrupted by either an error or a request for execution of a functional module via the EXECUTE control statement. In these two cases, control returns from XCSP to the executive monitor to determine what action is to be taken next. In all cases, the XCSP module remains in control and cycles through the read/process iteration for each control statement encountered. It is during this processing phase that the pre-stored set of control statements referenced by a CALL control statement is edited and processed before continuing with the next control statement. The edited form of the called control statement is written onto the data base and is available for subsequent retrieval. Thus editing is done only once at the first execution of the CALL statement. Any looping to re-execute the CALL statement will not cause redundant editing but only re-execution of the previously edited control statements.

When control returns from XCSP to XM, either error processing or functional module execution is indicated. If error processing is indicated, XM calls the error module XMERR to perform action according to procedures discussed in Section 1.10. Regardless of action, XMERR always returns to XM upon completion. If functional module execution is indicated, XM calls XFM to control the loading, execution, and clean-up processes. The functional module can make use of any and all services of the dynamic storage manager, the data base manager, and the executive utilities. The only restriction is that the functional module cannot terminate abnormally, but must return control to XFM and thus to XM. XFM will perform some clean-up procedures if the functional module has neglected to release or close dynamic storage areas or data base structures. Control then returns to XM.

When error processing or functional module execution has been completed, XM again calls the executive control statement processor module (XCSP) to continue executing the control statements.

In summary, the executive management system first performs bootstrap initialization and edits the primary input stream control statement set. Then it cycles among control statement processing, functional module execution, and error handling until completion.

1.7 DYNAMIC STORAGE MANAGEMENT

The ANOPP dynamic storage management system is a collection of modules that perform specific operations on the dynamic core areas discussed earlier. These modules are directly callable by the ANOPP executive and functional modules. Local dynamic storage and global dynamic storage are treated separately but equally by the modules of the dynamic storage management system. However, the modules of the executive management system do not treat them equally. The global dynamic storage area is initialized by the XBS module and never released during the rest of an ANOPP execution. Local dynamic storage on the other hand is initialized and released repeatedly by various executive modules and each functional module that makes use of it. For both types of storage, the remaining functions of dynamic storage management can be performed only during the time between initialization and release.

The basic function of the dynamic storage manager is to allocate and de-allocate blocks of storage within the initialized dynamic storage areas. Each block is located by an index with respect to the variable IX in a fixed common block /XANØPP/. This index is generically referred to as the IDX of the block. When a dynamic core block is assigned to a calling module, the index (IDX) and length (LEN) are returned to the module. The module can then reference any location within the block by addressing between the limits of IX (IDX) to IX (IDX+LEN-1). Alternatively, the module can pass the argument IX(IDX) and LEN to a submodule that receives them as an array A of length LEN and can address any location within the block from A(1) to A(LEN).

The initialization of a dynamic storage area consists of setting the boundaries with control words and declaring the remaining area to be one large free block. The size of the free block is reduced as reserved blocks are requested and assigned to calling modules. Eventually a reserved block will be freed by a calling module and it will be linked into a chain of free blocks along with the now reduced original free block. This process of reserving, reducing, freeing and chaining goes on until a request is made for more reserved space than is contained in any one of the individual blocks in the free chain. At this time a storage move takes place to consolidate all the fragmented free blocks into one large free block, unless storage has been locked at the request of a calling module. The IDX's of all relocated reserved blocks are updated accordingly. If the requested space is not available, the calling module is informed that the length of the block assigned is zero. In this case the calling module may free some blocks and try again, may request less space, or may request space in the other (local/global) storage area. When all else fails, the user can rerun the job with more field length.

1.8 DATA BASE MANAGEMENT

The ANOPP data base is a hierarchial structure from top to bottom and consists of:

LIBRARIES
UNITS
MEMBERS
RECORDS
ELEMENTS
WORDS

A library is a collection of units, a unit is a collection of members, a member is a collection of records, a record is a collection of elements, and an element is a collection of words.

Paralleling the hierarchial data base is a hierarchial data base management system consisting of directories, control statements, and subroutines that operate on individual parts of the data base or between adjoining parts. For example, the CREATE control statement establishes a new data unit while the UNLØAD control statement forms units or subsets of units into libraries. An overview of this parallel structure is given in Figure 1.

Units are equivalent to files in the host operating system. The Data Unit Directory (DUD) contains a table of correspondence between internal ANOPP data unit names and external host operating system file names. The collection of data units named in the DUD defines the current data base for ANOPP execution. It consists of those data units that have been created, attached, or loaded up to this point, and that have not yet been detached or purged. The physical external file for a data unit contains a unit header, a member directory of current members on the unit, and the actual members and records themselves. For an attached, created, or loaded data unit in the DUD, a copy of its unit header is kept as part of its entry in the DUD. An operating system I/O buffer in global dynamic core is also associated with a data unit's external file.

1.8.1 Member Manager

Below the level of unit is a member. A member and its substructures, records, elements, and words, are managed by a set of Member Manager Subroutines. These routines are callable from both executive and functional modules. A member is a collection of records

Data Base Structures	Directories	Control Statements	Subroutines
LIBRARY	LUT LFD LLT LUH LDR	LØAD, UNLØAD	

UNIT	DUD
	DMD
	DUH
		CREATE, PURGE
		ATTACH, DETACH
		UPDATE	

MEMBER	MCB DMH DUH RD AMD	-ADDR -COPY -CHANGE -ØMIT	MMØPWS MMØPWD MMØPRD MMCLØS MMREW
RECORD	RS	-INSERT -DELETE -QUIT	MMGETR MMPUTR MMSKIP MMPØSN
ELEMENT	FSI FST		MMGETE MMPUTE
WORDS			MMGETW MMPUTW

Figure 1. Data Base and Data Management Parallels

of the same format. Records are not formatted in the sense of format conversion as with FORTRAN coded records. Rather, the format indicates the structure of the records by giving the types of the record elements. Element types specify whether the data is integer, real, complex, single, double, or an alphanumeric string. The types are equivalenced to word lengths such as one word for an integer and four words for a complex double. Record reads and writes are really copying of binary data from/to external physical files to/from machine memory. The associated format provides a module writer with information for accessing individual record elements within sequential FORTRAN arrays.

When a data member resides on a physical external file, it contains a member head followed by the records of the member. The member header contains record format information as well as a record directory and subdirectory of member records relative to the beginning of member. When a data member is open for I/O, its name is included in an Active Member Directory (AMD), and a Member Control Block (MCB) is assigned in Global Dynamic Storage. These entries point to the DUD entry for the unit on which the member resides. The DUD entry points to the last operating system file buffer for the external file. All of these thread back to a NAME array that is used in every Member Manager call for action on the member.

The Member Manager routines provide capabilities to open, write, read, position, and close a member and its records. All calls to Member Manager subroutines provide a three word NAME array to indicate the data unit name and member name and to hold a pointer to the MCB. When a member is opened to read or write, a Member Control Block in dynamic storage is assigned and pointed to by the NAME(3) argument. The MCB points to the Active Member Directory entry for the open member and the Data Unit Directory entry for the unit on which the member resides. The Active Member Directory points both to the Data Unit Directory entry for the unit/member named and to the NAME argument supplied in the open call. The Data Unit Directory points to the External File Buffer (EFB) for the operating system file equivalenced to the data unit. See Figure 2, Diagram of Member Manager Directory Connections.

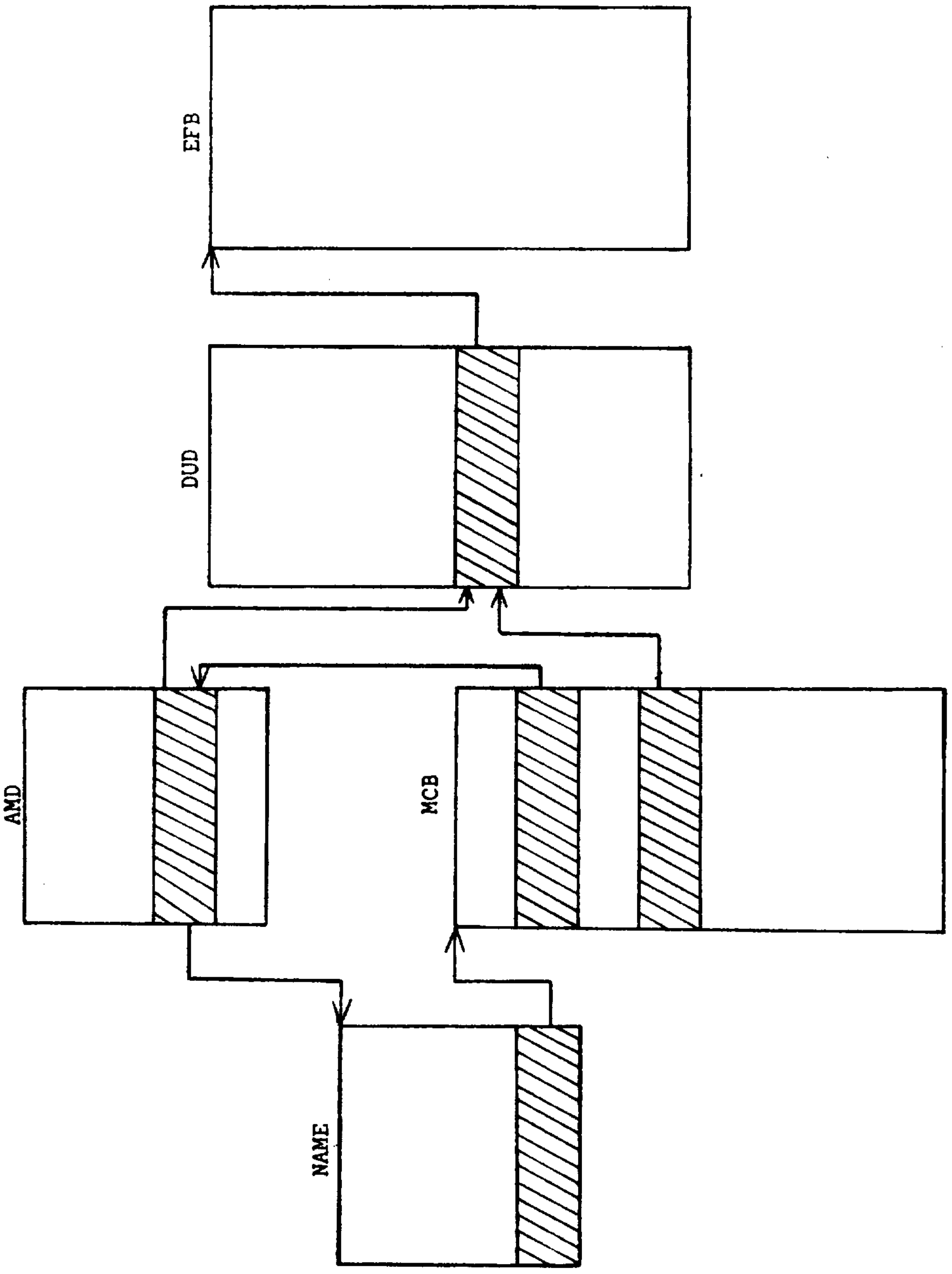


Figure 2. Diagram of Member Manager Directory Connections

All the records of a member occupy contiguous space on a data unit. Thus, when a member is open to write either sequentially or randomly, it often is necessary to actually write the records to a scratch unit until the member is closed. At that time all the records on the scratch unit are copied to the member's space on the real unit and the scratch unit is released. It is permissible to write records directly to a unit/member, but only one member at a time may be open for writing directly to the same unit.

When a member is opened to read the following actions take place. The unit name is found in the DUD and its Member Directory is read into core through the External File Buffer (EFB). If the member is found on the unit, then a Member Control Block is assigned and the Member Header is read into the MCB area. A member open to read entry is made in the Active Member Directory. With all these connections established at open time, the Member Manager is now ready for subsequent read requests to transfer data from the external file into a central memory record holding area. Records can be read in whole or in part with partial records specified by either word length or number of elements. Members can be rewound and records can be read sequentially or randomly with the additional capability to skip forwards or backwards over records or to position directly to a specified record. When closed, the MCB is released and if there are no other members currently open on the same data unit, the EFB is released also. The AMD entry is closed for reading, and if the member is not currently open to write also then the AMD entry is released. A member may be open to read and write concurrently since the member to be read must be an existing member in the current Member Directory for the data unit and the member open to write will go to a new place on the data unit and will not modify the Member Directory for the unit until the write process is closed. Since there is no re-write-in-place, read will continue to process the old version of a member while the new version is being written.

1.8.2 Table Manager

An ANOPP data table is a one record member. For this special class of members, the record holding area in dynamic storage into which the one record of the member is read is reserved and managed by the ANOPP Table Manager. The table name (unit/member) is held in a Data Table Directory along with a pointer to the table (record) area in global dynamic storage. Thus, these tables can remain in core under control of Table Manager during the execution of several functional modules. Like other units/members, the units/members whose one record constitutes a table cannot remain open after execution of a functional module. It is only the table data in the record holding area that remains in core under control of the Table Manager.

When a table is first opened, the unit/member is opened, the table record is read into core, and the unit/member is closed. The table name and a pointer to its core location is entered into the Data Table Directory. When the table is closed by the module that opened it, the table is logically closed in the DTD. A subsequent request to open the table will open it logically in core from the DTD. A request to close it will again close it logically. If a functional module opens a table with the intention of altering it, then it is rewritten to its real unit/member at the same time that it is logically closed in core.

The Data Table Directory holds a fixed number of tables in core -- some open, some closed. A request to open a table is first satisfied by opening the table if it is found in a search of the closed table chain in the Data Table Directory. If the table is not closed in core and available for re-opening, then an empty entry must be found in the DTD so that the unit/member/record for this table can be read into core and open member status can be entered into the DTD. If there are no free entries in the DTD, then one of the closed table entries will be released to make room for the new table entry. A subsequent request to open the table that was released will result in a fresh copy being read into core and re-entered in the DTD. If the DTD is full of open tables only, then no new tables can be entered and the job must be re-run with a new parameter value (NAETD) on the ANØPP control statement to initialize a DTD large enough to accommodate all the tables expected to be simultaneously open during the run.

1.9 UPDATE

An update capability has been provided for manipulating the ANOPP data base by reconfiguring the members of a unit and/or changing the records of a member. This capability has been provided at the ANOPP control statement level so that a user can control the outcome of an ANOPP execution by adjusting the data base prior to executing a functional module or unloading units of the data base to a sequential library.

The update capability is patterned after the structured organization of the data base. For modifying data units, there are record level directives to add, copy, omit, or change a member. For modifying data members there are record level directives to insert and/or delete records. This capability is non-destructive. In no case are the records or members of a unit rewritten on the same unit. In all cases update executes from an old data unit to a new data unit, or to a new data unit alone if no old unit exists. Update can operate in full or partial mode. In partial mode, only those members mentioned on update directives are processed from the old to the new data unit. In full mode, all members are processed.

One use of update is to reconfigure data units. This may be done to reduce wasted space on physical storage devices or to reorganize or combine members on a data unit in a manner more conducive to efficient execution by the intended ANOPP user. Another use is to create temporary data for use during an execution.

1.10 ERROR PROCESSING AND TERMINATION PHILOSOPHY

ANOPP may terminate either normally or abnormally. Normal termination occurs when the ENDCS control statement is processed. Abnormal termination occurs when a fatal error inhibits further meaningful execution is encountered.

Abnormal termination procedures are invoked whenever a fatal error is encountered by any executive module. Execution control is not passed back to the calling module by a return but instead a call is made directly to one of several error message modules. After printing an informative message, the message module calls the ANOPP abort module, XEXIT, to perform abnormal termination procedures and to terminate execution. Only executive modules have abort privilege. A functional module may never terminate ANOPP execution directly. However, abnormal termination may occur indirectly when a functional module is in core if an executive module, called to perform a service function, detects a fatal error. Abnormal termination will then occur as described.

Errors which do not inhibit further execution are called non-fatal and are detected by either executive modules or functional modules. All errors detected by a functional module are non-fatal to the executive system. If a module is able to correct the error situation and continue processing, no further action is required. If, however, the module is unable to successfully complete its function, return is made to the executive management system with the executive error parameter NERR set to .TRUE. A functional module must not terminate abnormally but must return control to the executive management system.

If an error occurs during either the initialize or edit phases of executive management, then execution continues and ANOPP is abnormally terminated upon completion of edit. Thus all errors in card image input are detected in one ANOPP execution. If an error occurs in later phases of processing control statements or execution of functional modules a return is made to the controller XM with NERR set as previously indicated. Subsequent action depends upon the system parameter JCON. Processing will continue with either the next control statement or the first encountered PRØCEED control statement.

STANDARDS

2.1 SCOPE

These standards define the requirements for preparing software for the Aircraft Noise Prediction Program (ANOPP). It is the intent of these standards to provide definition, design, coding, and documentation criteria for the achievement of a unity among ANOPP products.

These standards apply to all of ANOPP's standard software system. The standards as expressed in this publication encompass philosophy as well as techniques and conventions.

2.2 DESIGN

2.2.1 Module Standards

The Aircraft Noise Prediction Program will utilize the concepts of "composite design" for program structure. Composite design involves the construction of a program in terms of modular structure and module interfaces.

A module is a group of program statements that can receive input data, perform one or more transformations on that data, and return output data. The modules for ANOPP will have the following general characteristics.

1. The executable and comment statements for the module can be listed contiguously.
2. The statements are enclosed by identifiable boundaries; e.g., in FORTRAN, from a PROGRAM or SUBROUTINE card to an END card.
3. The statements are considered to be a discrete and identifiable entity that can be referenced from other parts of the program only by the module name or its single entry.
4. The module can be referenced from other parts of the program only by the module name or its single entry.
5. The module will have a single, common entry and a single, common exit.
6. The module can reference or CALL other modules, suspend its execution upon encountering the CALL statement, and resume execution with the next immediate statement.
7. All called modules must return to their caller at the statement immediately following the CALL statement.

A module has three attributes: function, logic, and interfaces. For a composite design, a module should be described by its functions; i.e., what happens when the module is called. This characteristic can be described as data flow through the program A functional description of a module should contain a verb, such as, "Find Largest Block". A module's logic describes the internal working or data flow within a module. Another attribute of a module, interconnection or interface, is concerned with module communication.

STANDARDS

ANOPP modules will be designed to be as functionally independent as possible with minimum interface. Reliability, ease of understanding, and ease of maintenance are the objectives of these standards.

Guidelines to be followed to achieve the standards of modularity for ANOPP are:

1. Simplicity - Use the simplest solution, design and/or interface that is possible.
2. Design efficiency - Design a module to solve the current problem efficiently; i.e., never design a module to do more that it is required to do.
3. Aligned control and effect - Align modules and decisions in modules so modules that are directly affected by a decision are beneath and controlled by the module containing the decision.
4. High strength - Maximize binding, the relationships among the elements of a module. For high strength or binding modules should:
   a. have all elements related to the performance of a single function,
   b. have a single entry and a single exit,
   c. have a function which is easy to describe,
   d. be independent from other modules,
   e. be unsusceptible to errors from complex design and coding,
   f. be usable by other programs, and
   g. be modifiable without affecting other modules.
5. Low interconnection - Minimize coupling, the relationships between modules. To attain the desired low interconnection or loose coupling, modules should:
   a. not directly reference other modules to:
      (1) modify a program statement,
      (2) refer to data in another module,
      (3) branch directly into another module, or
      (4) physically reside within another module;
   b. not pass control information to other modules;
   c. use only the following types of data interconnections:
      (1) argument lists,
      (2) common areas, and

(3) data base members.

6. Size - Limit the size of a module to 100 lines of executable source language statements. Clarity, simplicity, and understanding are related to module size.

2.2.2 Depth of Design

Program structuring involves an analysis of the problem, the flow of data through the problem, and the transformations that occur on that data. Equally, it involves identification and definition of modules to solve the problem. The phases of program structuring involve:

1. definition of the structure of the problem,
2. identification of the input and output in the problem,
3. identification of the points of entry and exit for data, and
4. reduction of the problem into a set of subordinate modules.

A graphic representation of a module and subordinate modules will result in a hierarchy of modules. The graphic representation is called a "hierarchy chart" and should be drawn to illustrate the problem's solution.

After the problem has been reduced into a set of subordinate modules, the process is repeated viewing each subordinate module as an independent problem that can be reduced into other subordinate modules. Some rules to follow are:

1. The entire structure should be constantly reviewed to take advantage of modules that are identical.
2. A module must be completely analyzed before its subordinates can be analyzed.
3. The order in which modules at the same level are analyzed and reduced is not important. It is not necessary to analyze a module and all of its subordinates before starting another module.

Conditions for terminating this iterative reduction process are:

1. When a module can be reduced no further into independent functional modules.
2. When further reduction of a module leads to the undesirable attributes of low

strength and high interconnections; i.e., low binding and high coupling.

3. When the logic of a module can be completely visualized; i.e., usually resulting in less than 100 executable statements.
4. When further reduction leads to highly specialized, unaligned, and inefficient sets.
5. When the resulting documentation (Chapin-style charts) can be drawn on two or less sheets of 8½" x 11" paper. (One page is the desirable objective.)

The final design phase of a module should follow these steps:

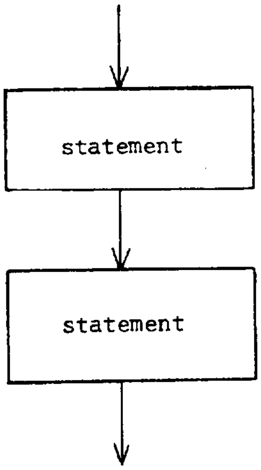
1. A Chapin chart should be drawn to illustrate the module's logic structure.
2. Documentation should be set down to define the module's purpose, inputs, outputs, local variables, functions, error conditions, control structures, data base structures, standards violations, and any additional explanatory remarks. This constitutes the first portion of the module's prologue and is intended to be included with the source statement listing.
3. A "walk through" of the module should be staged by the design team to insure workability.
4. Pseudo code reflecting the logic structure outlined by the Chapin chart should be completed. This completes the module's prologue.

2.2.3 Logic Structures

All of the ANOPP modules will be logically designed so the execution flow will be sequential from one logic structure to the next logic structure. ANOPP module design will use four basic logic structures. Coding of a logic structure may require more than one executable source statement. No matter how complex the particular structure, upon completion of its execution, the next structure will be executed. This sequential flow logic is characteristic of "top-down" design. The four logic structures are defined with traditional graphics in the following paragraphs for educational value and comparison with the ANOPP standard Chapin charts.

2.2.3.1 Simple Sequence Structure

2.2.3.1.1 Logical Flow Diagram

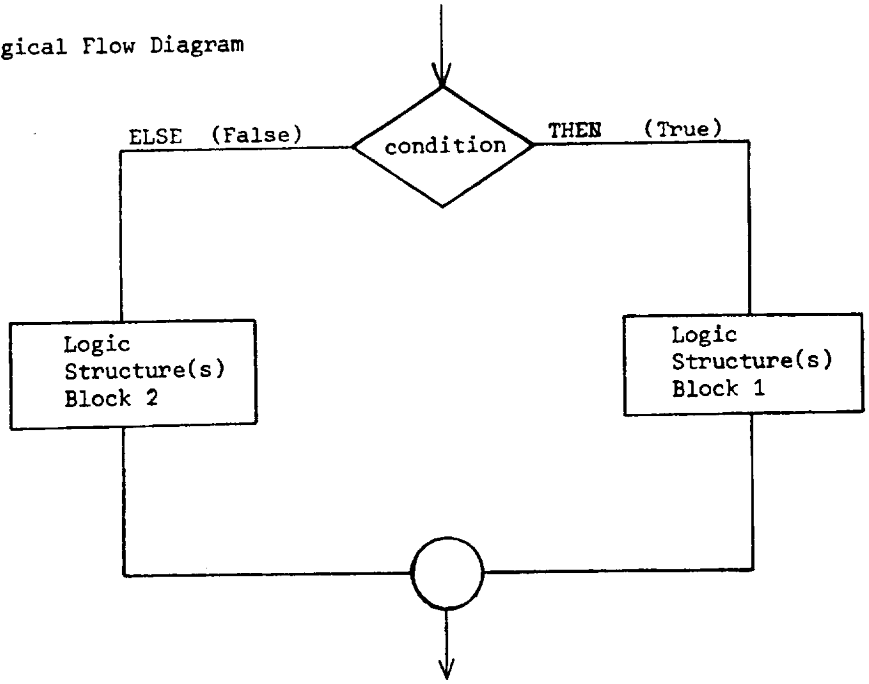


2.2.3.1.2 Description

Each statement within the structure is simple in that it performs one basic function; e.g., an assignment of an evaluated expression to a variable or a call to another module (or subordinate).

2.2.3.2 IF THEN/ELSE Structure

2.2.3.2.1 Logical Flow Diagram

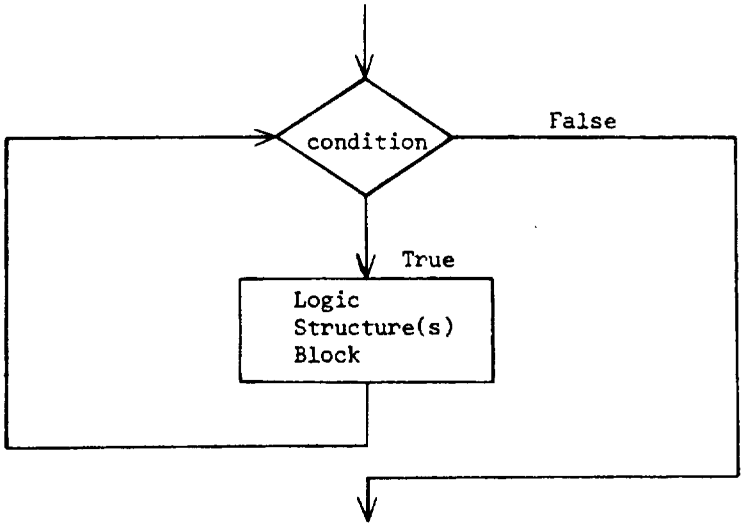


2.2.3.2.2 Description

The IF THEN/ELSE structure describes the flow sequence that occurs when there are two blocks of logic structure(s) and only one block should be executed according to a decision criteria or condition. The condition is a simple or a complex logical expression which has a value of True or False. If upon execution the condition is True, the logic structure(s) of Block 1 (the THEN path) is(are) executed. If the condition is False, the logic structure(s) of Block 2 (the ELSE path) is(are) executed. When the last logic structure in the chosen path has been executed, control goes to the next logic structure or statement following the IF THEN/ELSE structure. This logic structure adheres to the "top-down" design in that the common entry is the point at which the condition is tested and the common exit leads to the next logic structure.

2.2.3.3 DO WHILE/DO UNTIL Structure

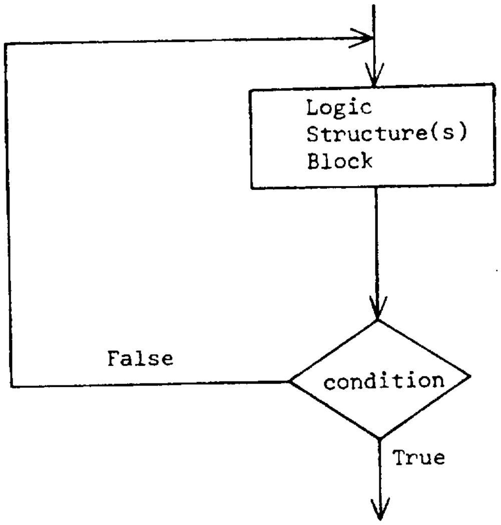
2.2.3.3.1 DO WHILE Logical Flow Diagram



2.2.3.3.2 DO WHILE Description

A block of logic structure(s) which may or may not be executed one or more times depending on a given condition is described by the DO WHILE loop structure. The loop structure adheres to "top-down" design in that the loop is entered at one point and flow within the loop progresses to one common exit point; i.e., the point at which the condition is tested. The condition is a simple or complex logical expression which has a value of True or False. The condition is tested at the beginning of the loop, before the execution of the logic structure(s) block, and if True, the logic structure(s) is(are) executed. When execution of the logic structure(s) is(are) complete, control returns to the beginning of the loop and the condition is tested again. Looping continues until the condition is False; control then passes to the next logic structure following the DO WHILE structure.

2.2.3.3.3 DO UNTIL Logical Flow Diagram



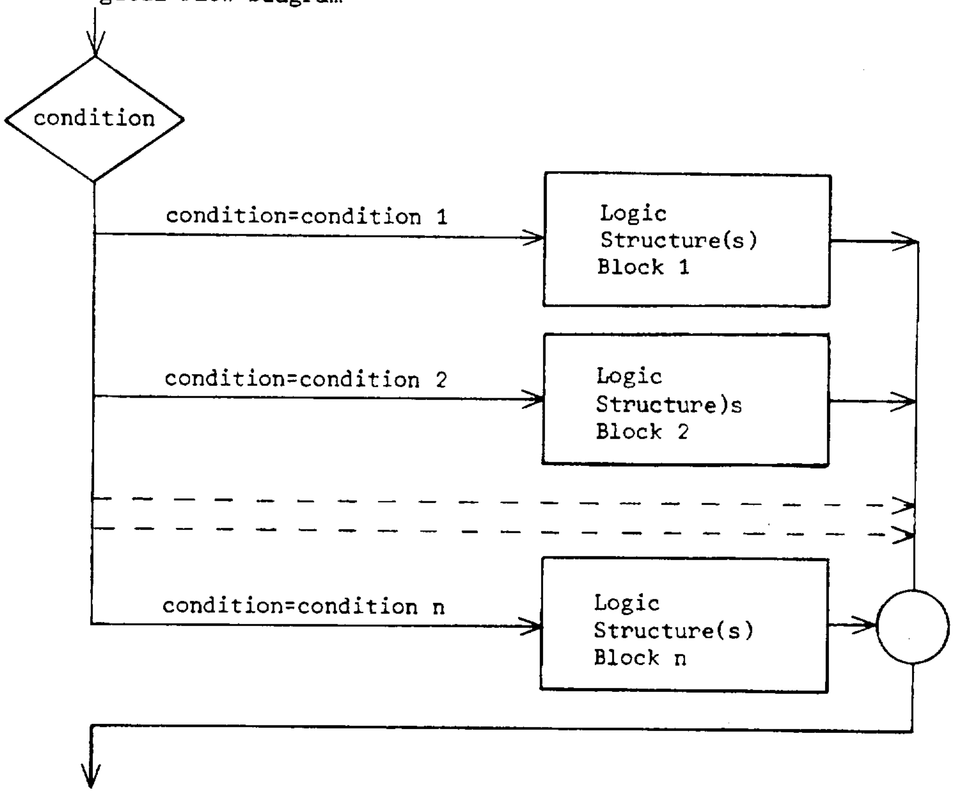
2.2.3.3.4 DO UNTIL Description

A block of logic structure(s) which will be executed one or more times depending on a given condition is described by the DO UNTIL loop structure. The structure adheres to the "top-down" design in that the loop is entered at one point and flow within the loop progresses to one common exit point; i.e., the point at which the condition is tested.

The condition is a simple or complex logical expression which has a value of True or False. The condition is tested at the end of the loop, after the execution of the logic structure(s) block, and if False the logic structure(s) block is executed again. This continues until the condition tested is True; control then passes to the next logic structure following the DO UNTIL structure.

2.2.3.4 CASE Structure

2.2.3.4.1 Logical Flow Diagram



2.2.3.4.2 Description

The CASE logic structure describes the flow sequence that develops when there are two or more blocks of logic structures, only one of which will be executed according to a given condition or decision criteria. The condition when evaluated must have a resulting value identical to one and only one of the conditions given (i.e., condition 1, condition 2, ..., condition n). Control will pass to the beginning of the block identified by the matching condition. Upon completion of the chosen block, control will pass to the next logic structure following the CASE structure. The CASE structure adheres to the "top-down" design in that the structure is entered at one point, the test condition point, and after execution of the chosen block, control passes to one common exit and the next logic structure.

2.2.4 Design Documentation

The design phase should result in a description of the structure of the program with descriptions of the module and intermodule interfaces. For the ANOPP project, the design phase will result in (1) hierarchy charts of the areas of the program, (2) Chapin charts with external specifications (module prologue) for each module depicted on the hierarchy charts, and (3) pseudo code.

2.2.4.1 Hierarchy Charts

A hierarchy chart will depict the results of the composite analysis process (top-down reasoning - structural process). This creative process is necessary to arrive at the modular logic and involves the analysis of the problem, the flow of data through the problem, and subdivision of the problem into modules that will perform transformations on the data.

A hierarchy chart depicts each module, the level of the module (order), and the lines of communication for the module. In its optimal form, a hierarchy chart should be contained on one page (Figure 1). As an area can have several levels of modules, it may be necessary to place a module with its subsequent levels on additional pages; however, all subordinate modules on the same level should be placed on the same page. In the example illustrated in Figures 2 and 3, five levels of modules are necessary. The submodules of the module "Control to Next Level" (Level 3) could not be depicted on the same page and were subsequently placed on an additional page. (Note the asterisk in the upper right corner of the "Control to Next Level" box. This indicates that this module is expanded as a separate hierarchy)

Each module will have a short title, descriptive of its function, and a name that is used to reference the module. Module names should be descriptive of the function.

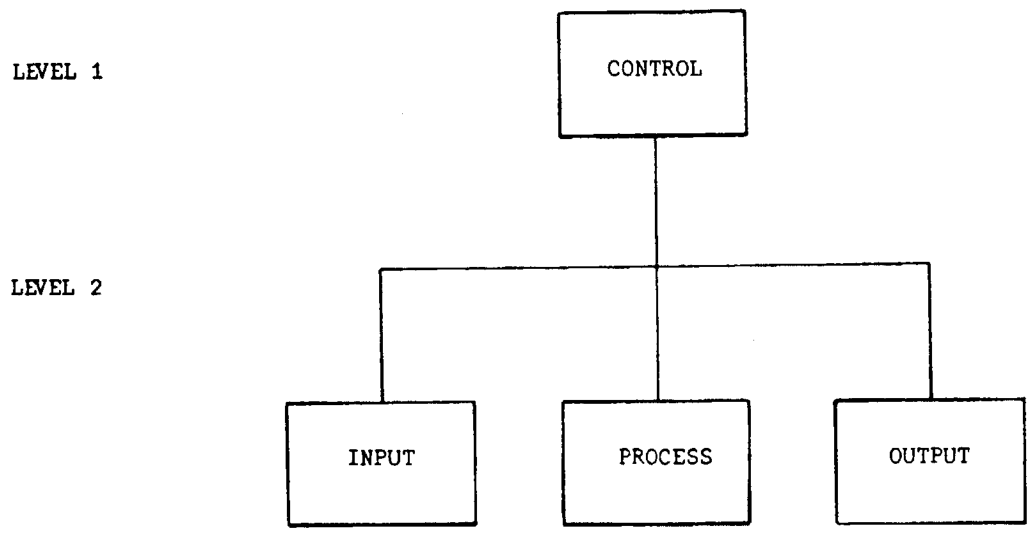


Figure 1. Hierarchy Chart: Basic Form

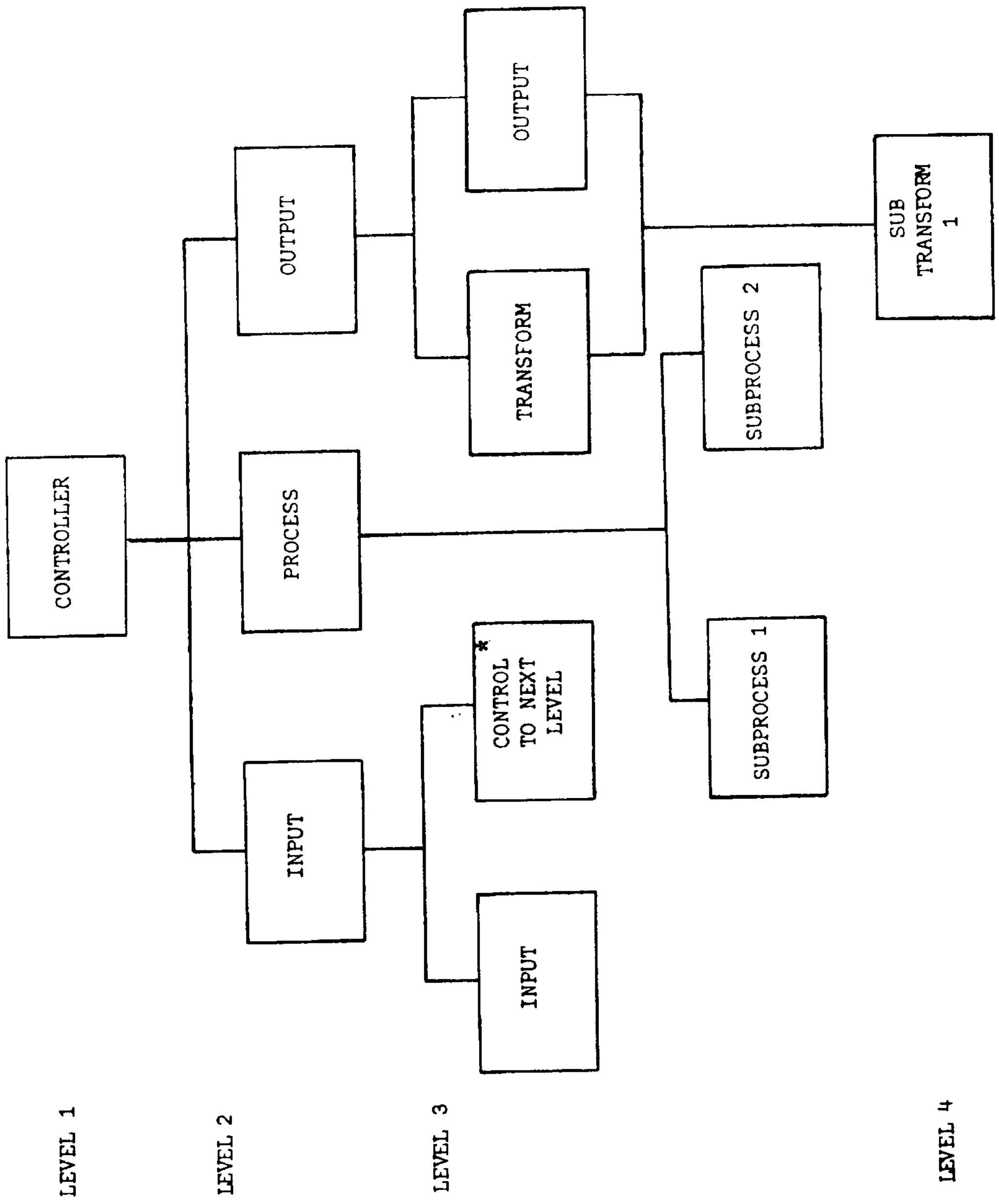


Figure 2. Hierarchy Chart: Basic Form

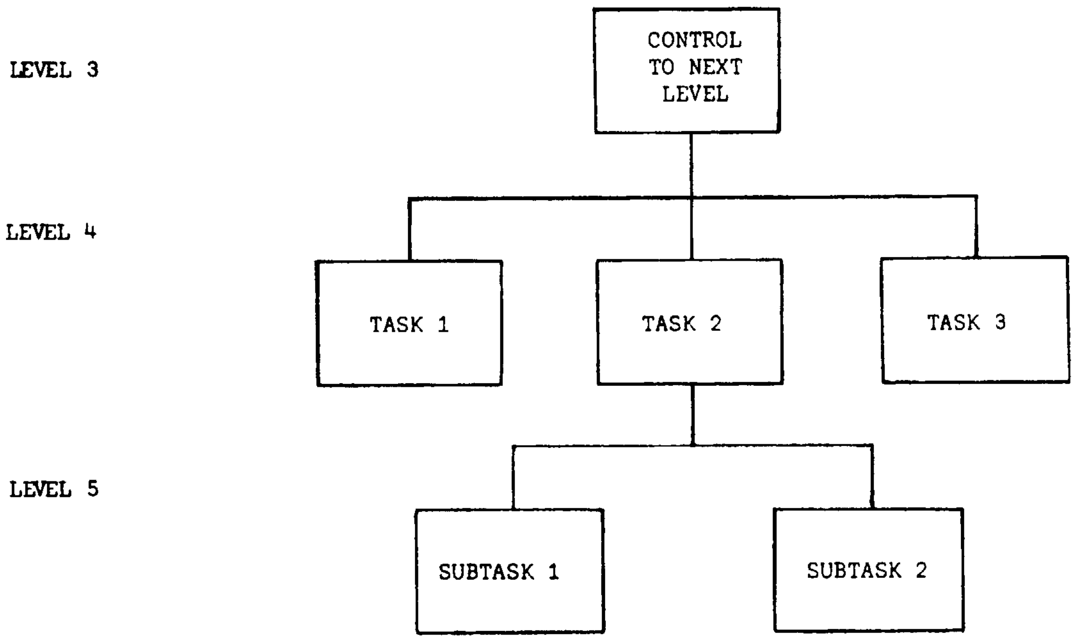
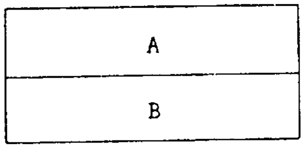


Figure 3. Haerarchy Chart: Basic Form

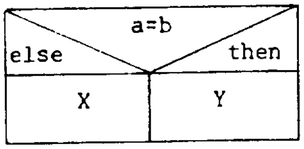
2.2.4.2 Chapin Charts

A Chapin chart will be drawn for each module depicted on the hierarchy chart. The drawn Chapin chart will detail the internal logic of the module using simple control structures. In a structured program module, any program function can be performed using one of four control structures. The Chapin forms for these structures are:

1. Simple sequence

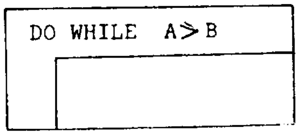


2. IF THEN/ELSE

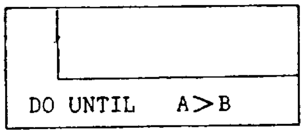


3. Repetition

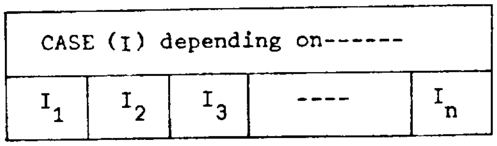
DO WHILE



DO UNTIL



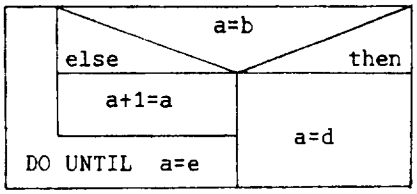
4. CASE

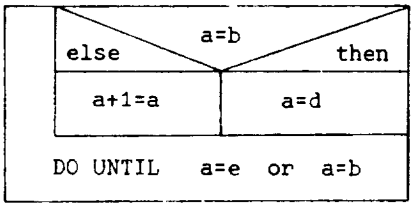


CASE I is really a generalization of the selection function (IF THEN/ELSE) from a two-valued to a multi-valued operation.

Any kind of processing, any combination of decisions, and any sort of logic can be accomodated with one of these control structures or a combination of these control structures. Each structure is characterized by a single point of transfer of control into the structure and a single point of transfer out of the structure. The control structures can be nested and still retain this characteristic.

A tricky situation, prevalent in current practice, arises when a designer desires to terminate a repetition block upon encountering a specific condition. If this termination is diagrammed as illustrated below, it violates the single entry/single exit principle of structured programming. However, equivalent logic is produced by using a multi-valued condition in the classical structure as shown below.

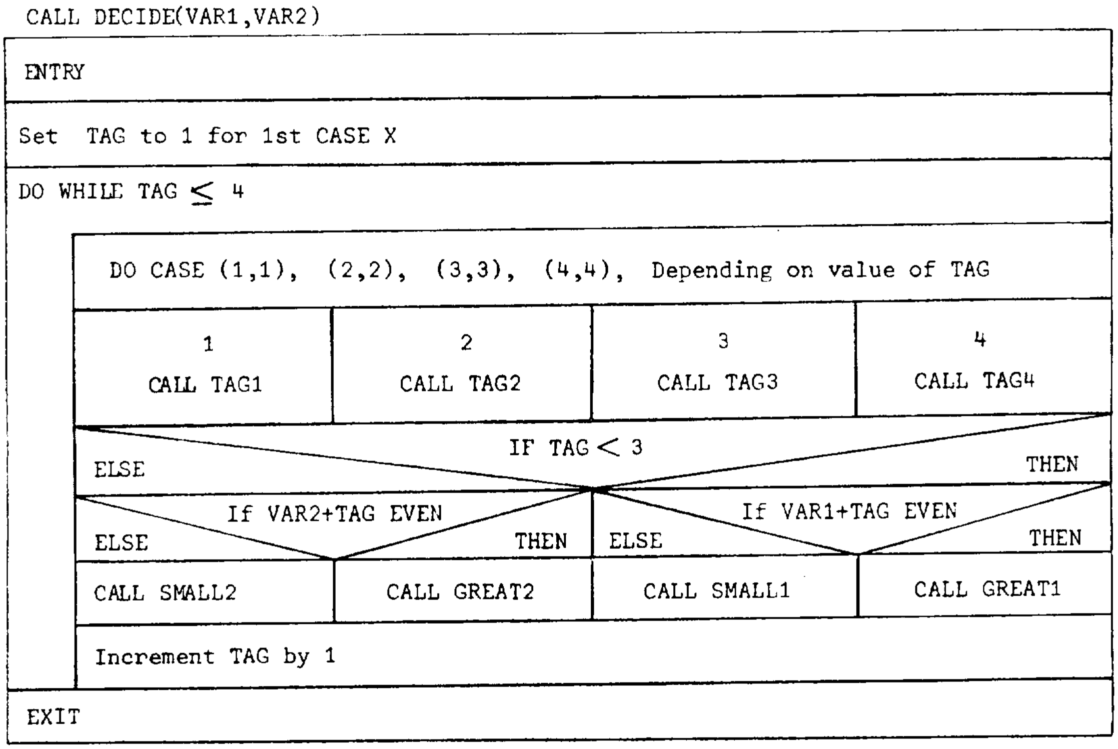




A program utilizing these control structures tends to have no statement labels. (The actual implementation of these structures in a non-structured language like FORTRAN will require the use of statement labels. See Section 2.3 - CODING.) Utilizing these control structures in a top-down design eliminates arbitrary and capricious branching in a module and results in a more precise flow of data.

The hand drawn Chapin charts (Figure 4) will be generated in the design process for formal "walk through" reviews of the structured design. The purpose of the review will be to uncover flaws in the design. The Chapin chart will enable the reviewers to examine the entire logic of the module.

In addition to the use of the restricted control structures, the Chapin charts will also contain other attributes for ease of understanding. The chart will contain the title and name of the module. The module name should be descriptive of the function performed by the module and is the name that is used to reference the module (for example, in a CALL statement).



Purpose - To initialize TAG areas and build tables.

Date - Designed/9/30/75, JD - Coded/10/15/75, MP

Functions - Call individual TAG areas in sequence. On the first two passes, build a small or large type one table first depending upon the value of the input parameter VAR1. On the next two passes, build a small or large type two table first depending upon the value of the input parameter VAR2.

Inputs - VAR1 = 1 if Great Table 1 is to be built first
VAR1 = 2 if Small Table 1 is to be built first
VAR2 = 1 if Great Table 2 is to be built first
VAR2 = 2 if Small Table 2 is to be built first

Outputs - None

Figure 4. Typical Chapin chart with external specifications.

The language for the Chapin chart should not be cryptic to the point that only the designer understands the logic. Neither should it be so wordy that it can't fit in the box.

The use of the IF THEN/ELSE control structure will sometimes result in a do-nothing or NULL statement from the question. It is preferred that the NULL statement be designed and implemented from the ELSE path of the question.

Module lengths should be limited to a manageable size. No firm rule can exist for size; however, the tendency is to have between 10 and 100 lines of executable statements. With this size, a single entry, single exit, and no arbitrary jumps to other parts of the program, there is little need for page-turning or holding several places which must be referenced constantly.

The function performed by the module should be described in a single sentence followed by an expanded description, if necessary. The expanded description can be a narrative description, tables, etc., and should be easily adaptable to card format for inclusion in the programming documentation.

There should be a precise description of all input and output data for the module. It should include all parameters, any physical order, size, type, and range of valid values. A full description of module interconnections is necessary as it will usually affect any calling module. Any external effects should be explained, e.g., the reading of a tape or printing.

The module name, functional description, input and output description, and external effects will be called the module's external specifications. For design, these items will be placed on the Chapin chart and/or additional pages if necessary. These items will be prepared to be carried onto the program listing as the first half of a module's prologue.

2.2.4.3 Pseudo Code

After the design "walk through" and approval, the Chapin chart will be converted into indented control structure pseudo code in punched card format. This will constitute the second half of the module's prologue.

DESIGN

Pseudo code, English phrases derived from the Chapin chart, describes the flow of the control structure. The simple sequence is a statement. The IF THEN/ELSE structure is divided into three parts: (1) IF is usually a one line question; (2) THEN is a statement to be executed if the answer is true; and (3) ELSE is a statement if the answer is false. The THEN statement and/or ELSE statement can be followed by other questions and/or statements. The DO UNTIL, DO WHILE, and CASE are statements.

The pseudo code acts as a bridge between the design and coding phases. It is a transformation of the highly graphic, parallel vision, Chapin charts into a form similar to the top-down, straight line, final source code. As such, there are several guidelines for converting the Chapin charts to pseudo code.

1. All decisions which alter the simple sequence flow of the program will be shown. If, during coding, a FORTRAN flow altering statement is introduced, it must be reflected in the pseudo code.
2. All FORTRAN CALL statements must be reflected as a simple sequence statement in pseudo code.
3. Each FORTRAN statement which is a simple sequence type does not require a matching statement in the pseudo code if it is part of a group of FORTRAN statements which performs a common pseudo code statement.

   Example:

PSEUDO CODE	FORTRAN STATEMENT
Calculate X, Y, Z coordinates	X = ____________
	Y = ____________
	Z = ____________

4. The control statements IF, DO WHILE, DO UNTIL, and CASE always denote additional statements will follow. Each of these control statements will use an appropriate END statement to denote the end of a particular set. The END statement will be indented the same number of columns as its subject. The END statements are ENDIF, ENDDO, and ENDCASE.
5. The pseudo code will be written in a format with strict indentation in each

group and subgroup of statements for ease of understanding and clarity. See Section 2.3.1 - Source Code Documentation for specific rules.

2.3 CODING

To ensure ease of understanding, maintaining, and interchanging of ANOPP code, certain standards will be imposed. These standards encompass both documentary comments and specific language statements. It is recommended that any exceptions to standards be employed only within the bounds of a specific module and not be allowed to couple with other modules.

2.3.1 Source Code Documentation

Comment statements in any programming language are both source code and documentation. Thus, various kinds of descriptive information which would normally appear in publishable programming documentation can be captured as comments in the source code also. For ANOPP, design and documentary information will be brought together and placed in the program source listing. This information will be contained in a special module prologue section at the beginning of a routine and in regular comment statements interspersed among the executable statements of a routine. The prologue section should explain the purpose and functioning of a routine as well as the flow of control within the routine. If any coding standard is violated, it must be noted in the prologue and, as an additional comment, in the executable statements. The in-line comments are supplementary in nature and should explain special cases or values and other non-obvious implementations.

The first line of a routine is the program header, be it PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA, or IDENT. The module prologue will appear immediately following the program header and preceding any other lines of source code. The source code and optional comments will follow the prologue.

The module prologue contains both descriptive information and a pseudo code translation of the module's Chapin chart. The definition of prologue contents and format is given below. An example of a prologue is illustrated in Figure 1.

ANOPP PROLOGUE FORMAT

```
COLUMN
0   0    1 1 1 1
1   5    0 2 4 6

***
*        PURPOSE - short description of subprogram function (1 - 2 sentences)
*
*        AUTHOR - initials (level number such as L01/00/00)
*
*        INPUT
*          ARGUMENTS
*            Name_1 - description
*            .
*            .
*            .
*            Name_n - description
*          OTHER
*            /common block name/
*              Name_1, ...,Name_n - described in subprogram name
*            or
*            /common block name/
*              Name_1 - description
*              .
*              .
*              .
*              Name_n - description
*              or
*              Verbal description if required
*              (For common variable, only those applicable to the module should be
*               listed.  The full description of each variable is required for major
*               modules.  However, for sub-modules, a reference to where the description
*               can be found is sufficient.)
*
*        OUTPUT
*          ARGUMENTS - same as for INPUT
*          OTHER - same as for INPUT
*
*        LOCAL VARIABLES
*          Name_1 - description
*          .
*          .
*          .
*          Name_n - description
*
*        FUNCTIONS
*          1.  Function_1
*          .
*          .
*          .
*          n.  Function_n
*
*        CONTROL STRUCTURES
*          Description of control structures or reference to description in another
*          subprogram or published manual.  Control Structures include core tables,
*          directories, control blocks, etc.
*
```

```
*        DATA BASE STRUCTURES
*          Description of control structures or reference to description in another
*          subprogram or published manual.  Data Base Structures are unit, member, and
*          record structures.
*
*        SUBPROGRAMS CALLED
*          Subprogram_1, ..., Subprogram_n
*
*        ERRORS
*          NON-FATAL
*            1.  Condition_1 (error message class and number)
*            .
*            .
*            .
*            n.  Condition_n (error message class and number)
*          FATAL
*            same as for NON-FATAL
*
*        STANDARDS VIOLATIONS
*          1.  Short description
*          .
*          .
*          .
*          n.  Short description
*
*        REMARKS
*          Additional comments
***
*   ENTRY
*        Pseudo Statement (in columns 10, 15, 20, etc.) - simple sequences and the
*        following sets of key words should be aligned in order within a set: IF, THEN,
*        ELSE, ENDIF; DO CASE, CASE, ENDCASE; DO WHILE, DO UNTIL, ENDDO.  Subsequent
*        substructures should be indented 5 spaces.
*   EXIT
***
```

The headings PURPOSE, AUTHOR, INPUT, OUTPUT, FUNCTIONS, ERRORS, AND SUBPROGRAMS CALLED will be mandatory. If a mandatory heading is not applicable, "None" should be indicated after the heading and all subheadings omitted (e.g. INPUT - NONE). If a mandatory heading is applicable, all subheadings under it must be specified. The headings LOCAL VARIABLES, CONTROL STRUCTURES, DATA BASE STRUCTURES, STANDARDS VIOLATIONS, and REMARKS are optional and, if not applicable, should be omitted.

The statements in the pseudo code should be labeled with statement numbers where appropriate. These numbers should be in numerically ascending sequence from the top down. Corresponding FORTRAN statements in the source code should be similarly numbered in the same top-down sequence.

```
      INTEGER FUNCTION NWDTYP( ITYPE )
***
*
*        PURPOSE - DETERMINE THE NUMBER OF WORDS REQUIRED FOR A DATA
*                  TYPE GIVEN ITS ANOPP INTEGER TYPE CODE.
*
*        AUTHOR -  SSS(L01/00/00)
*
*        INPUTS
*          ARGUMENTS
*            ITYPE      ANOPP INTEGER TYPE CODE
*          OTHER
*            /XCVT/
*              NDTCL, NMH, NCPW - DEFINED IN /XCVTBD/
*
*        OUTPUT
*          INTEGER FUNCTION VALUE OF NUMBER OF WORDS IN FIELD
*
*        FUNCTIONS
*          1.  DETERMINE THE NUMBER OF WORDS IN A FIELD GIVEN ITS TYPE
*              CODE
*          2.  VALIDATE TYPE CODE
*              INVALID CODES ARE ZERO AND OUT OF RANGE STRING VALUE
*
*        DATA STRUCTURES
*              SEE ANOPP PROGRAMMERS REFERENCE MANUAL FOR FULL
*              DESCRIPTION
*          1.  ANOPP DATA TYPES TABLE
*
*        SUBPROGRAMS CALLED
*          XUFMSG
*
*        ERRORS
*          NON-FATAL - NONE
*          FATAL
*            1.  INVALID ANOPP TYPE CODE
*                XUFMSG ERROR MESSAGE NUMBER 4
***
*    ENTRY
*         IF TYPE CODE IS BETWEEN 1 AND 10
*         THEN FIND NUMBER OF WORDS IN ANOPP DATA TYPES TABLE
*      10 ELSE IF FIELD ILLEGAL (TYPE CODE GREATER THAN 20)
*              THEN COMPUTE NUMBER OF WORDS FROM CODE
*      20      ELSE IF FIELD CHARACTER STRING (TYPE CODE BETWEEN -1 AND
*                   -132)
*                   THEN COMPUTE NUMBER OF WORDS FROM CODE
*      30           ELSE ILLEGAL TYPE CODE
*                        ABORT WITH MESSAGE
*      40           ENDIF
*      50      ENDIF
*      60 ENDIF
*    EXIT
***
```

Figure 1. Prologue and executable statement listing.

CODING

```
C
      LOGICAL NERR
C
      COMMON /XCVT/                NERR            ,NEXPND
     1       ,NDT                 ,NDTCL(12,3)     ,NBPW
     2       ,NCPW                ,NBPC            ,NMH
     3       ,NTNAME              ,NTMAX           ,NTCUR
     4       ,NTENT               ,NTSTRT          ,NT3USD
     5       ,NT3FRE              ,NT3OTR          ,NT3STR
     6       ,IWR                 ,IRD             ,LENGL
     7       ,NWPCI               ,NMCPW
      IF((( ITYPE.LT.1 ).OR.( ITYPE.GT.10 )) GO TO 10
      NWDTYP = NDTCL(ITYPE,3)
      GO TO 60
   10 IF( ITYPE.LE.20 ) GO TO 20
      NWDTYP = (ITYPE-21+NCPW)/NCPW
      GO TO 50
   20 IF((( ITYPE.GE.0 ).OR.( ITYPE.LT.-NMH )) GO TO 30
      NWDTYP = (-ITYPE+NCPW-1)/NCPW
      GO TO 40
   30 CALL XUFMSG( 4, 6HNWDTYP, 5HITYPE, ITYPE )
C     THE CALL ABOVE SHOULD ABORT
C     THE STOP BELOW INHIBITS ILLOGICAL EXECUTION
      STOP
   40 CONTINUE
   50 CONTINUE
   60 CONTINUE
      RETURN
      END
```

Figure 1. Prologue and executable statement listing. (Continued)

Section 2.2.4.3 described the design conversion from Chapin chart to pseudo code. Section 2.3.2 will describe the translation from pseudo code to FORTRAN source code. Thus, the pseudo code in the prologue is a highly visible bridge between the module as designed and the module as coded. Capturing this documentation in the source code will simplify the tasks of understanding, testing, and maintaining a module's code.

2.3.2 FORTRAN Language Standards

The requirements of ANOPP will impose certain restrictions on the use of the normal FORTRAN language for two reasons. First, the requirement for machine independence demands the use of a FORTRAN subset that operates compatibly on several manufacturer's computers. Second, the set of requirements for structured programming and its attendant simple logic structures demand several implementation algorithms since FORTRAN is not a structured programming language.

2.3.2.1 Machine Independence

FORTRAN, a high level compiler language, is relatively machine-independent. Even so, standard FORTRAN (ANSI X3.9-1966) has not been implemented by the same or different manufacturers to be completely independent of machine architecture. However, a fundamental precept of ANOPP development is to minimize implementation and conversion problems on the major third generation scientific computers (CDC CYBER series, IBM 360/370 series, UNIVAC 1100 series). To this end, ANOPP code will conform to ANSI standards as defined in the FORTRAN Extended Version 4 Reference Manual subject to the restrictions listed below.

NON-ANSI constructions (indicated by shaded areas in the reference manual) must not be employed. The following are standards that are to be followed to maximize machine independence.

1. The magnitude of an integer constant or variable may not be greater than $2^{31}-1$.
2. Subscripted variables should contain no more than 3 subscripts.
3. Array variables must be referenced with explicit subscripts, e.g., A(1) = 0, not A = 0.

4. A CØNTINUE statement requires a FORTRAN statement number.

5. The PAUSE statement is not to be used.

6. The NAMELIST statement is not to be used.

7. Implied DO's in DATA statements are not allowed. An array name without subscripts is allowed although it is an ANSI violation.

8. The last statement of a DO loop may not be a logical IF statement.

9. BLOCK DATA subprograms may contain only type (e.g., REAL, INTEGER), DIMENSION, COMMON, and DATA statements.

10. All variables containing Hollerith data should be limited to eight characters left-justified and blank-filled. The forms nL and nR should not be used for Hollerith data. The form nH should be used. When using Ai format specification, i must not exceed 8. A3, A6, A8 are valid; A10 is invalid.

11. Packed fields within a computer word should not be used.

12. Octal (O or B) or Hex (Z) in DATA or FORMAT statements may not be used.

13. Specification statements should precede any executable statement.

14. The order of specification statements should be as follows:

    COMPLEX

    DOUBLE PRECISION

    REAL

    INTEGER

    LOGICAL

    EXTERNAL

    DIMENSION

    COMMON

    EQUIVALENCE

    DATA

15. The variables in a COMMON block should be ordered as follows: complex, double precision, real, integer, and logical.

16. Variables stored as single precision cannot be referenced as double precision variables (via the FORTRAN EQUIVALENCE statement) because of the different

internal word storage format for single and double precision words.

17. Caution must be exercised to insure that types (REAL, INTEGER, etc.) of FORTRAN functions agree in the function subprogram and in the calling program. This agreement between types is necessary for machines (e.g., IBM 360) in which REAL and INTEGER values of FORTRAN functions are returned in different registers.

18. No attempt to extend the length of arrays through the EQUIVALENCE statement should be made.

19. Caution must be exercised when using the EQUIVALENCE statement. Optimizing compilers do not guarantee that the values used for the equivalenced variables will be the expected value. Hence, EQUIVALENCE should be used only between variables which have non-intersecting use spans in a program. Storage and retrieval of a variable value is not necessarily in the order given by FORTRAN source.

20. Multiple entry routines and routines with nonstandard returns are not to be used.

21. There must be agreement with respect to the number of arguments and the type of each argument in the argument list of a calling program and the called subroutine.

22. Only the carriage control characters "1" and "blank" may be used to control printer spacing. No spacing or suppression of spacing characters may be used.

23. Modification of the length of an explicit type declaration (e.g., REAL*8) is not allowed.

24. Deck (or member) names for subroutines should be six or less characters and should agree with the primary entry point names. Deck names for Block Data subprograms should end with the characters "BD".

25. FUNCTION subprograms whose type is not implicit must be typed in the FUNCTION statement. For example, use

```
          DOUBLE PRECISION FUNCTION ABC( X )
```

and not

```
FUNCTION ABC( X )
DOUBLE PRECISION ABC
```

26. The name of a FUNCTION subprogram must appear somewhere within the subprogram.

27. All subscripted variables appearing in EQUIVALENCE statements must be subscripted, e.g., use EQUIVALENCE (A(1), X(1)) instead of EQUIVALENCE (A,X).

28. DO loop indices may not be greater than $2^{17} - 1$ (131,071).

29. Logical operations are permitted on non-logical variables only using supplied functions IAND, IOR, ICOMPL, IXOR.

30. Subscripts may not contain subscripted variables.

31. Actual subroutine parameters that are changed by the called subroutine must have unique locations.

    Example: CALL SUB(A, A) where SUB is as follows:

    ```
    SUBROUTINE SUB( C, D )
    C = 10
    RETURN
    END
    ```

    is not allowed.

32. No DATA statements for variables in common blocks outside BLOCK DATA programs will be used.

33. Blank common will not be used.

34. ENCODE, DECODE, or similar installation or machine dependent routines will not be used.

35. Branching into the range of a DO statement is not allowed.

36. It is preferred that the use of constant numbers for referencing or indexing tables be restricted.

2.3.2.2 Structured FORTRAN

FORTRAN is not a structured programming language. FORTRAN syntax does not directly include the logic constructs defined in Section 2.2.3 and Section 2.2.4 of this document. However, several implementation algorithms can be defined to permit adherence to the concept of structured programming.

2.3.2.2.1 Simple Sequence

FORTRAN syntax permits easy implementation of the simple sequence structure. There is no need for statement numbers within the sequence except for format statements that do not alter the flow of execution. Entry to the sequence may require a statement number if it is entered as the result of a previous branching structure.

Example:

```
Pseudo Code                       FORTRAN Code

GET X                             READ 100, X
COMPUTE SQUARE ROOT OF X          SX  = SQRT( X )
PUT RESULT                        PRINT 100, SX
```

2.3.2.2.2 IF THEN/ELSE

FORTRAN syntax does not include an IF THEN/ELSE structure. However, various combinations of Arithmetic If, Logical If, Go To, and Continue statements can provide the two-branch logic desired. The rules for their use are as follows:

1. The THEN path must precede the ELSE path in top-down order in both the pseudo code and the FORTRAN code.
2. The ENDIF statement must be represented by a numbered Continue statement.
3. The Arithmetic If statement with two of the three branches equal is the preferred implementation.
4. The Logical If must be used with logical variables or multiple relational expressions.
5. The Logical If must be implemented with a .NOT. condition and a Go To the ELSE path.

Example 1: Arithmetic If

```
Pseudo Code                           FORTRAN Code

  IF ANGLE .LE. 180                     IF( PI - THETA ) 2, 1, 1
                                                  or
                                        IF( THETA - PI ) 1, 1, 2
1 THEN COMPUTE SIN( ANGLE )           1 SINTH = SIN( THETA )
                                        GO TO 3
2 ELSE COMPUTE - SIN( 180 - ANGLE )   2 SINTH = -SIN( PI - THETA )
3 ENDIF                               3 CONTINUE
```

Example 2: Logical If

Pseudo Code

```
  IF SWITCH IS TRUE
  THEN CALL SUBA

1 ELSE CALL SUBC
2 ENDIF
```

FORTRAN Code

```
  IF( .NOT.( SWITCH ) ) GO TO 1
  CALL SUBA
  GO TO 2
1 CALL SUBC
2 CONTINUE
```

Example 3: ELSE path null

Pseudo Code

```
  IF X .LT. ZERO


1 THEN X = ABS( X )
  ELSE NULL
2 ENDIF
```

FORTRAN Code

```
  IF( X ) 1,2,2           (preferred)
      or
  IF( .NOT.( X.LT.0 ) ) GO TO 2
1 X = ABS( X )

2 CONTINUE
```

2.3.2.2.3 CASE

DO CASE (condition 1, statement 1) ....(condition n, statement n) on test variable

The DO CASE structure can be implemented with a variety of programming techniques employing a combination of Go To, Computed Go To, Logical If, Arithmetic If, and Continue statements. In all cases the ENDCASE statement must be represented by a CONTINUE statement with a statement number. Four basic examples are illustrated.

Example 1: Integer Test Variable

Pseudo Code

```
  DO CASE (1,1) (2,2) (3,3) (4,4)
          (5,5) on I
    1 Process Control Statement 1

    2 Process Control Statement 2

    3 Process Control Statement 3

    4 Process Control Statement 4

    5 Process Control Statement 5
6 ENDCASE
```

FORTRAN Code

```
  GO TO (1, 2, 3, 4, 5) I

1 CALL PRCS1
  GO TO 6
2 CALL PRCS2
  GO TO 6
3 CALL PRCS3
  GO TO 6
4 CALL PRCS4
  GO TO 6
5 CALL PRCS5
6 CONTINUE
```

Example 2: Arithmetic If

```
Pseudo Code                             FORTRAN Code

  DO CASE (-,1) (0,2) (+,3) on X          IF ( X ) 1, 2, 3
    1 SX = SQRT( -X )                   1 SX = SQRT( -X )
                                          GO TO 4
    2 SX = 0                            2 SX = 0
                                          GO TO 4
    3 SX = SQRT( x )                    3 SX = SQRT( X )
4 ENDCASE                               4 CONTINUE
```

Example 3: Logical If with an executable statement

```
Pseudo Code                             FORTRAN Code

  DO CASE ("A",1) ("B",2) ("C",3)
          ("D",4) on NAME
    1 CALL SUBA                         1 IF( NAME.EQ.1HA ) CALL SUBA
    2 CALL SUBB                         2 IF( NAME.EQ.1HB ) CALL SUBB
    3 CALL SUBC                         3 IF( NAME.EQ.1HC ) CALL SUBC
    4 CALL SUBD                         4 IF( NAME.EQ.1HD ) CALL SUBD
5 ENDCASE                               5 CONTINUE
```

Example 4: Logical If with Go To

```
Pseudo Code                             FORTRAN Code

  DO CASE ("A",1) ("B",2) ("C",3)         IF( NAME.EQ.1HA ) GO TO 1
          ("D",4) on NAME                 IF( NAME.EQ.1HB ) GO TO 2
                                          IF( NAME.EQ.1HC ) GO TO 3
                                          IF( NAME.EQ.1HD ) GO TO 4
                                          GO TO 5
    1 CALL SUBA                         1 CALL SUBA
                                          GO TO 5
    2 CALL SUBB                         2 CALL SUBB
                                          GO TO 5
    3 CALL SUBC                         3 CALL SUBC
                                          GO TO 5
    4 CALL SUBD                         4 CALL SUBD
5 ENDCASE                               5 CONTINUE
```

2.3.2.2.4 DO WHILE (condition) and DO UNTIL (condition)

The DO WHILE structure implies that condition testing is done before the sequence of operations is performed. The DO UNTIL structure implies that the sequence of operations is performed at least once before condition testing is done. These structures can be implemented in FORTRAN with various combinations of Arithmetic If, Logical If, Go To, Continue, and Do statements. The possible variations are too myriad to enumerate specifically. However, the following standards will promote adherence to the spirit of top-down structured programming.

CODING

1. A Do statement can be used for a DO UNTIL with an incrementing condition.
2. If a Do statement is used for a DO WHILE, then the condition must be tested as the first statement inside the Do or as the statement immediately preceding the Do if it is an incrementing condition with variables instead of constants.
3. All FORTRAN Do loops must end with a numbered CONTINUE statement.

2.3.2.2.5 CONTINUE

A CONTINUE statement is used if required to insure that the DO WHILE or DO UNTIL will stand alone (i.e., not depend on the previous or next pseudo code structure). A labeled CONTINUE statement can therefore appear in the FORTRAN code with a label number not appearing in the Pseudo code. The label numbers should, of course, appear sequentially in the FORTRAN code. Examples 1, 3, and 4 show a labeled CONTINUE not shown in the Pseudo code but required for implementation. Example 2 shows the absence of any CONTINUE statement.

Example 1

Pseudo Code

```
   DO UNTIL ( TABLE(I) = NAME FOR
            I = 1 to TABLELENGTH)
       .....
       .....

10 ENDDO
```

FORTRAN Code

```
   DO 9 I = 1, LTAB

   .....
   .....
 9 CONTINUE
10 CONTINUE
```

Example 2

Pseudo Code

```
10 DO UNTIL A EQUALS B
   ......
   ENDDO
```

FORTRAN Code

```
10 ......
   ......
   IF( N.NE.B ) GO TO 10
```

Example 3

Pseudo Code

```
   DO WHILE ( I.LE.K FOR I = J
             TO K)
   ......
   ......

10 ENDDO
```

FORTRAN Code

```
   IF( J.GT.K ) GO TO 10
   DO 9 I = J, K
   ......
   ......
 9 CONTINUE
10 CONTINUE
```

Example 4

```
Pseudo Code                          FORTRAN Code

 9 DO WHILE ( A.EQ.B .OR. C           9 IF( .NOT.( A.EQ.B .OR. C.GT.D ) )
           .GT.D )                     *GO TO 10
   .....                                .....
   .....                                .....
                                        GO TO 9
10 ENDDO                             10 CONTINUE
```

2.3.3 Assembly Language Standards

There is no unique assembly language that is compatible across several manufacturers' computers. The assembly language standards for ANOPP, then, fall into two categories: (1) general requests to adhere to the spirit of structured programming and (2) specific interface requirements between FORTRAN and assembly language subroutines for a given machine.

2.3.3.1 General Rules

The following general rules will promote clarity and understanding while adhering to the spirit of structured programming.

1. Each routine shall have a module prologue as described in Section 2.3.1.
2. Multiple entry points and non-standard returns are not allowed.
3. Self-modifying code is not permitted. Instructions can change data only, they cannot change other instructions.
4. Assembly code must follow the same top-down order as the pseudo code.
5. Liberal use of comments is recommended for clarity and understanding.
6. The prologue's pseudo code should be repeated in appropriate comment fields of assembly statements.
7. Local macro definitions should be found after the prologue at the beginning of a routine.
8. System macros should briefly be explained and a document reference cited.

2.3.3.2 COMPASS/FORTRAN Interface

Several conventions must be observed when FORTRAN and COMPASS subroutines are intermixed. For FORTRAN Extended, the conventions are explained in the Fortran Extended Version 4 Reference Manual. A brief list follows:

1. Every COMPASS subroutine shall have:
   - a. IDENT and END cards beginning in column 11;
   - b. A trace word of the form VFD 42/name, 18/entry address;
   - c. An entry point of the form name DATA 0;
   - d. An entry point name agreeing with the deck name on the IDENT statement;
   - e. Register A0 saved on entry and restored upon exit.
2. Function subroutines shall return single precision values in register X6; double precision and complex values are returned in registers X6 and X7.
3. Subroutine and function calls are performed by a return jump sequence with trace information and argument addresses passed through an argument list. The form is as follows:

```
          SA1     ARGLIST
        + RJ      =X external subprogram name
        - VFD     12/line number, 18/trace word address
          ...
  ARGLIST VFD     60/ARG1 address
          VFD     60/ARG2 address
          ...
          VFD     60/ARGM address
          VFD     60/0 end of argument address list
```

Sample parts of a COMPASS subroutine are shown in Figure 2.

```
          IDENT SAMPLE

***
*
*P
* R
*  O
*   L
*    O
*     G
*      U
*       E
*
***

          ENTRY SAMPLE
TRACE     VFD    42/OLSAMPLE, 18/SAMPLE      Trace word
TEMPAO    DATA   0                           Holding location for AO
EXIT      SA1    TEMPAO                      Restore AO
          SAO    X1
SAMPLE    DATA   0                           Entry point
          SX6    AO                          Save AO
          SA6    TEMPAO
          SAO    A1                          Save input argument list
          ...                                address
          ...
          SA1    ALIST                       Call SUB(A,B,C)
   +      RJ     =XSUB
   -      VFD    12/*-TRACE, 18/TRACE
          ...
          SA1    AO+1                        Fetch 2nd argument
          SA1    X1
          ...
          ...
          EQ     EXIT          Restore AO and return through entry point
          ...
ALIST     VFD    60/A          Address of A
          VFD    60/B          Address of B
          VFD    60/c          Address of C
          VFD    60/O          End of argument list
   A      DATA   0             Storage for A
   B      DATA   0             Storage for B
   C      DATA   0             Storage for C
          ...
          END
```

Figure 2. Sample parts of COMPASS routine.

2.4 TESTS

Testing is the activity that takes coded pseudo and FORTRAN statements and removes compiler statement errors, input formatting errors, output formatting errors, and program structural and logic errors. The testing activity is composed of (1) desk checking, (2) component testing, (3) integration testing, and (4) system testing.

2.4.1 Desk Checking

Upon completion of coding of the FORTRAN or assembly statements for a module, the product should be reviewed with the Chapin chart and the pseudo code for completeness and accuracy. The module is compiled and all compiler generated errors are removed to obtain an error-free compilation.

2.4.2 Component Testing

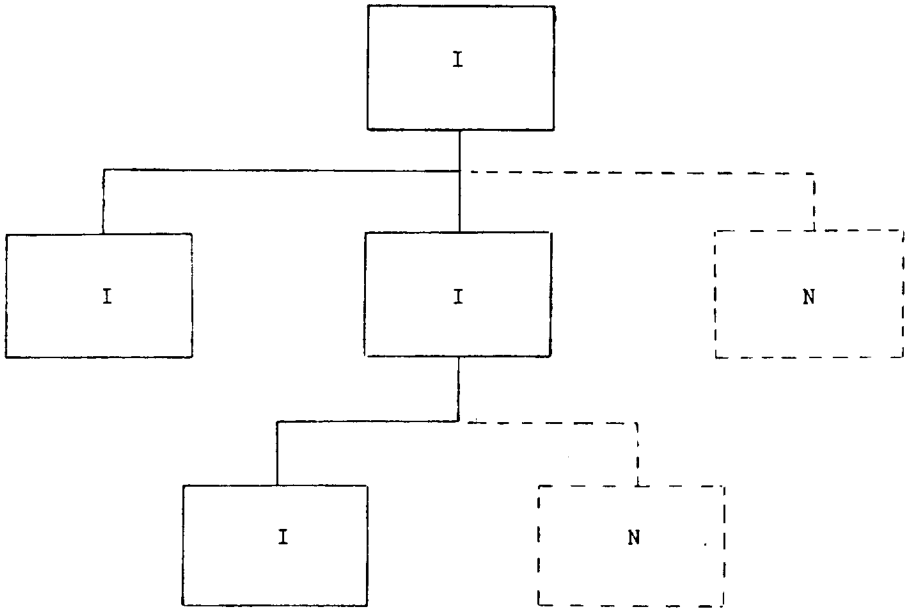
Component or isolation testing is that activity which takes a module, exercises it through its full range of inputs and outputs, and evaluates its performance for any necessary correction. Each and every path of a module must be exercised during component testing. Stubs for other modules that are referenced must be generated to allow a smooth run to completion.

Standards for component testing will produce tests that will:

1. exercise typical error free cases,
2. exercise error free worst case,
3. produce each error code,
4. produce variations of errors,
5. vary all system parameters affecting the module for the above runs, and
6. vary user options affecting the module for the above runs.

2.4.3 Integration Testing

After component testing, the module is integrated into the program. In the figure below, all modules designated with I are integrated and the modules designated with N are not integrated. A module is never integrated into the program unless it is subordinate to a previously integrated module.



When a module is integrated into the program, integration tests will be performed. Integration and testing is the activity wihch places the tested module into the program and exercises the module. The module is exercised as thoroughly as possible for interaction with other modules. The tests should check for:

1. typical case with no errors,
2. worst case with no errors (checking for efficiency and any designed limits),
3. each error code,
4. all possible errors in one entry,
5. various typical cases,
6. various system parameters that affect the module, and
7. various user options that affect the module.

The test cases and results of integration tests will be documented and saved for later use. These test cases can be used to determine if the program is operating as designed.

2.4.4 System Tests

System testing is the activity of exercising the program utilizing all inputs in various combinations. According to the concepts of structured programming and top-down module integration, as the last module is integrated and tested, testing of the entire program will be complete. However, tests will be conducted for:

1. the ANOPP control statement stream with no errors and composed of
   a. simple sequences,
   b. various typical combinations, and
   c. worst case combinations;
2. the ANOPP control statement stream with errors, such as,
   a. meaningless input,
   b. no input,
   c, stacked sets of input, and
   d. errors;
3. all system parameters for
   a. typical settings,
   b. special cases,
   c. worst cases, and
   d. illegal values.

2.5 PUBLISHABLE DOCUMENTATION

2.5.1 Types of Publishable Documentation

A program of ANOPP's magnitude requires clear and complete documentation to be of value to users and programmers. Engineers and users must understand the available prediction capabilities and know how to formulate a problem and obtain a solution. Programmers must know how to install, modify, and add to the system. Such documentation will be provided in four separate, indexed, stand-alone documents:

1. Programmer's Manual,
2. Theoretical Manual,
3. User's Manual, and
4. Demonstration Problem Manual.

2.5.1.1 Programmer's Manual

The Programmer's Manual will contain all coding information and specifications for the Aircraft Noise Prediction Program. It will be written for use by programmers to install, execute, modify, and add modules to the program. As such, it will contain:

1. a detailed introduction that will describe the concepts and functions of ANOPP;
2. the standards for design, coding, testing, and program documentation to insure compatibility and ease of maintenance;
3. description of data and tables;
4. executive, data management, utility, and functional module descriptions;
5. instructions for installation and operation;
6. instructions for modification and addition of modules to the system;
7. support program descriptions; and
8. easily updated index.

2.5.1.2 Theoretical Manual

The Theoretical (or methods) Manual will provide a concise mathematical description of the methods employed in the computational or functional modules. It will describe the analytical or empirical methods and will outline the methods of solution, including all implicit and explicit assumptions, limits of use and limits of accuracy. References to published material should be included in the text and the index. A user can refer to this manual to determine the engineering and mathematical methods that are available in the program.

2.5.1.3 User's Manual

The User's Manual will be structured to accommodate the needs of different levels of users. A user will employ this manual to formulate problems and anticipate results. The manual will provide instructions and descriptions for the preparation of problem data and explain how to invoke the various options provided for problem solution. The User's Manual will thoroughly explain the Executive Control Language and general related capabilities.

2.5.1.4 Demonstration Problem Manual

The Demonstration Problem Manual will contain detailed descriptions of sample problem input and solutions. This manual will be utilized for user education and system validation. It will be beneficial to the engineer user and his programming staff.

2.5.2 Publishable Manual Preparation

2.5.2.1 General

ANOPP documentation will be typed on 10" x 13" mats that will be supplied by ANOPO. These mats will be reduced to 90 percent of their original size during the printing process.

Documents will be prepared using magnetic cards compatible with the equipment available to ANOPO. The equipment available to ANOPO is an IBM MAG Card II Typewriter, System Model No. 6616; specifications: dual pitch word processing system.

The magnetic cards of the ANOPP manuals will be furnished to ANOPO with an index to facilitate cataloging and filing the cards.

PUBLISHABLE DOCUMENTATION

Computer printout used in the manuals must be clear and sharp. To ensure this, the unlined side of the computer paper should be used and a new ribbon should be inserted on the printer.

To change camera-ready manuscripts, correction tape is preferred over mortising (i.e., cutting and pasting). Also, for one-letter corrections, a chalk-like substance may be used. ERASURES AND OPAQUE WHITE CORRECTION FLUID ARE NOT ACCEPTABLE.

Minor modifications to pages of published ANOPP documentation will be communicated to ANOPO by means of the Documentation Change Report (DCR).

The general format of the manuals will be:

1. Front Cover
2. Inside Cover Page
3. Preface
4. Table of Contents
5. Page Status Log
6. Text Body
7. References
8. Index
9. Back Cover

2.5.2.2 Spacing

Double-spacing will be used except where groups of a few single-spaced lines separated by double-spacing for the groups is more desirable for clarity or appearance.

Paragraphs will be indented 5 spaces and will be separated from each other by 2½ lines. A line associated with an unnumbered, underlined explanatory heading will be indented 5 spaces.

Section and subsection titles will be separated from the text (above and below the title and from each other) by 3 lines.

2.5.2.3 Section Numbering

Major sections, i.e., those with one number, will be typed with all uppercase letters and will be identified with a decimal classification, i.e., one number followed by a period, as follows:

12. DOCUMENTATION

Major subsections, i.e., those with two numbers, will be typed with all uppercase letters and will be identified with a decimal classification and two numbers, as follows:

12.2 MANUAL PREPARATION

Minor subsections, i.e., those with three numbers, will be typed with initial capitals, underlined, and identified with a decimal classification and three numbers, as follows:

12.2.5 <u>Contents of Manual</u>

Further subdivision of minor subsections will be typed with initial capitals, will not be underlined, and will be identified with a decimal classification and four or more numbers, as follows:

12.2.5.3 Text Printing

2.5.2.4 Page Numbering and Running Headings

Major subsections will begin at the top of an odd-numbered page. (Odd-numbered pages will be printed on the right and even-numbered pages will be printed on the left.) Other units such as Data and Table Descriptions should begin at the top of a page where clarity or convenience of use is thereby improved. In the case of large major subsections, minor subsections may begin at the top of the next page.

Page numbers will be centered at the bottom of each page. The number will indicate the major subsection identifier and page number within the subsection, separated by a hyphen. Examples are:

6.1-1

6.1-2

6.1-3

The numbers of pages changed at a later date will use the format of major subsection identifier, hyphen, page number followed by the date of the change (mm/dd/yy), as follows:

6.1-1	original page
6.1-2 (12/09/75)	changed page

Pages inserted at a later date will be identified by the major subsection identifier, hyphen, page number, decimal and number followed by the date of the insertion in the form (mm/dd/yy), as follows:

6.1-1	original page
6.1-2 (12/09/75)	changed page
6.1-2.1 (12/09/75)	added page
6.1-2.2 (12/09/75)	added page
6.1-3	original page

If an odd number of pages is to be inserted, one blank page with a running header and a page number should be added to ensure consistency. Such a blank page will contain the following sentence: THIS PAGE HAS BEEN LEFT BLANK INTENTIONALLY.

Pages inserted at a later date between pages of insertions made subsequent to the original issue will be identified by the major subsection identification number, hyphen, page number, decimal, added page, decimal, inserted page and date as follows:

6.1-1	original page
6.1-2 (12/09/75)	changed page
6.1-2.1 (12/09/75)	added page
6.1-2.1.1 (01/10/76)	inserted page
6.1-2.1.2 (01/10/76)	inserted page
6.1-2.2 (12/09/75)	added page
6.1-3	original page

Running headers, in capitals, will be centered at the top of each page. The major section name will be used as the running header on EVEN-NUMBERED PAGES, and the major subsection name will be used as the running header on ODD-NUMBERED PAGES. There is one major exception to this rule: for the first page in every major subsection, the major section name will be used as the running header. Care must be exercised in determining running headers for pages to be inserted. For example, the page to be inserted between 6.1-3 and 6.1-4 is 6.1-3.1 and is considered to be an even-numbered page for the purpose of determining the running header. The reason for this is that 6.1-3.1, being printed on the back of 6.1-3 (an odd-numbered page), is in effect an even-numbered page. A blank "odd-numbered" page, 6.1-3.2, with subsection running header and page number must be typed to insure consistency. This blank page will contain the following sentence: THIS PAGE HAS BEEN LEFT BLANK INTENTIONALLY.

2.5.2.5 Equations

Equations will be numbered consecutively beginning with 1 for the first equation in each major subsection. References to equations outside a major subsection must refer to both the equation number and the major subsection number, i.e., See Section 5.6, Equation 12. If the reference is to another manual, the manual name (Theoretical Manual, User's Manual, Programmer's Manual) will be given, i.e., See ANOPP Theoretical Manual, Section 5.6, Equation 12.

Equations will be centered on the line and separated from the text by three blank lines. Equations will be punctuated as part of the text and will be identified at the right-hand margin with its Arabic numeral equation number in parenthesis.

When a group of equations appears in succession without text between them, the longest equation will be centered and the equal signs of the remaining equations will be aligned with the equal sign of the longest equation.

The transpose operator for a matrix should be placed outside the brackets (i.e., $[A]^T$ is correct; $[A^T]$ is incorrect). The same rule applies to the inverse operation (i.e., $[A]^{-1}$ is correct; $[A^{-1}]$ is incorrect). All subscripts for matrices will be lowercase letters (i.e., K_{99} , K_{fs}).

All plus and minus signs in equations will be preceded and followed by one space. There will be two spaces before and after all equal signs in equations. There will be no spaces between parenthetical expressions. For example:

(A)(B) + (D) = C

Equations inserted at a later date will be numbered as decimal parts of the preceding equation. For example, two equations inserted between Equation 5 and Equation 6 will be designated by Equation 5.1 and Equation 5.2, respectively. For more complicated cases, follow the rules given in Section 2.5.2.4 of this document.

2.5.2.6 Tables, Figures, and References

Tables and figures will be numbered consecutively beginning with the first table or figure in each major subsection. References to tables and figures outside a major subsection must refer to both the table or figure number and the major subsection number, i.e., See Section 5.6, Table 2. If the reference is to another manual, the manual name (ANOPP Theoretical Manual, ANOPP User's Manual, ANOPP Programmer's Manual) will be given, i.e., See ANOPP User's Manual, Section 5.2, Table 2.

Table titles will be typed with initial capitals. Periods will follow the Arabic table number and the end of the complete title as follows:

Table 3. This Is an Example of a Table Title.

Figure captions will be typed in lower case letters except for the first letter of the first word. Periods will follow the Arabic figure number and the end of the complete caption as follows:

Figure 4. This is an example of a figure caption.

Single line captions will be centered under the figure, and multiple-line captions will be left-justified with the last line centered.

References will be listed at the end of each major section and will be numbered consecutively beginning with 1 for the first reference in each major section.

Tables, figures, and references inserted at a later date will be designated by a number followed by a decimal and a number. For example, two figures inserted between

Figure 2 and Figure 3 will be designated Figure 2.1 and Figure 2.2, respectively.

The words Equation, Figure, Reference, Section, and Table will be spelled out with initial capitals when used either in the text or in a caption. The associated Arabic numeral will not be enclosed in parentheses.

2.5.2.7 Capitalization

Data block names, module names, Data card names, entry point names, and FORTRAN variable names will all be capitalized and the letter O will be slashed (Ø).

Care should be exercised in the use of initial capitals. Formal type names should be capitalized throughout the manuals.

2.5.2.8 Punctuation

Commas will be used to separate the elements in a series and a comma will be placed before the final conjunction.

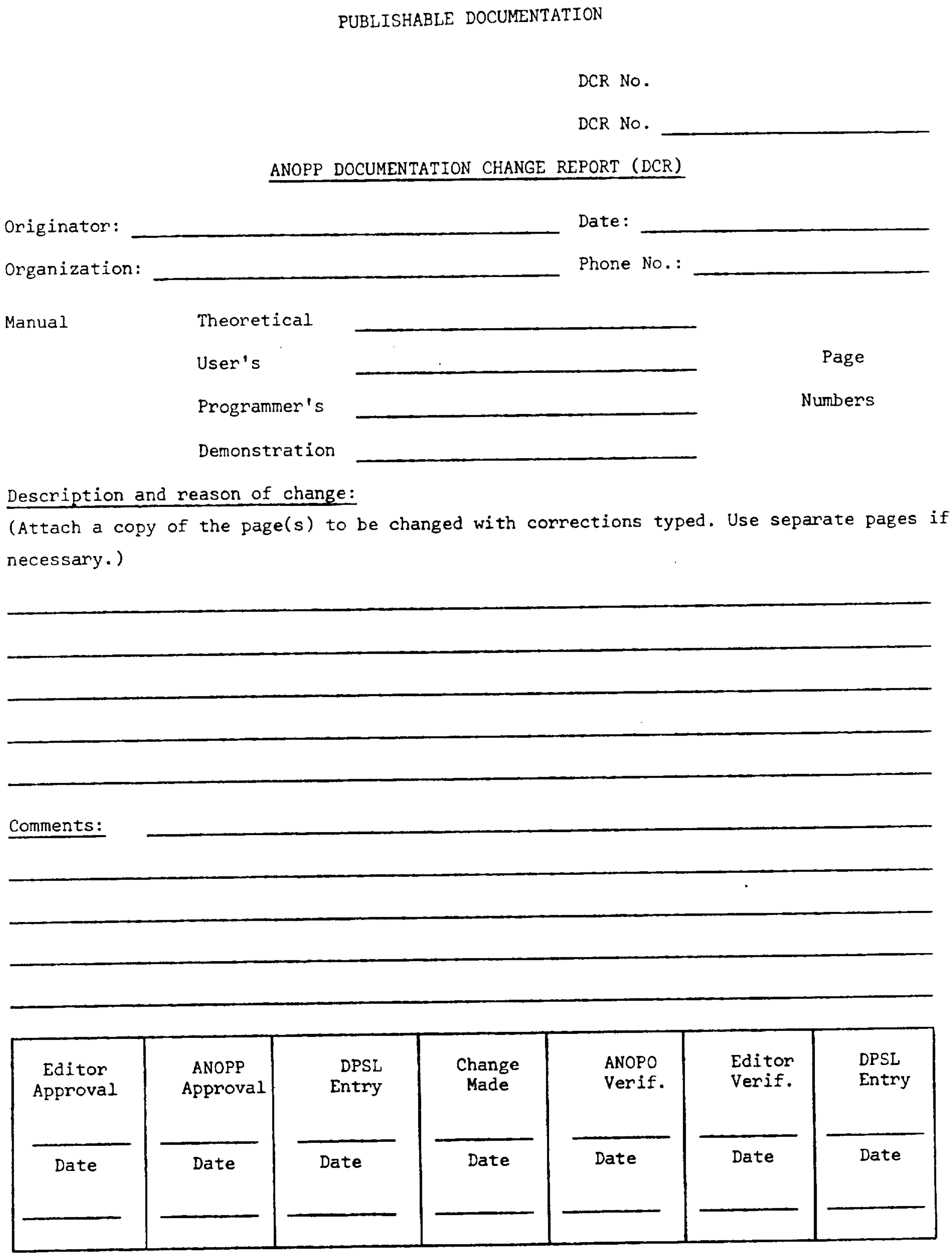
For punctuation of potential executable statements, punctuation characters should be representative of the actual coded statement.

2.5.3 Changes to Baseline Manuals

The four ANOPP manuals (Theoretical Manual, User's Manual, Programmer's Manual, and Demonstration Problem Manual), delivered to ANOPO via paper, called mats, and IBM MAG Card II compatible magnetic cards, constitute baseline documents. When information in a baseline document is added, deleted, or changed, a formal written update to the baseline document is required.

To initiate a change to a baseline document, a Documentation Change Report (DCR) is required and must be submitted to the maintenance organization. A DCR is shown in Figure 1.

The report will be reviewed by the maintenance organization and/or ANOPO for appropriateness and extent of change. Changes to manuals can affect software. The results of the reviews will consist of comments, required changes, and suggested changes as well as



PUBLISHABLE DOCUMENTATION

DCR No.

DCR No. ____________

ANOPP DOCUMENTATION CHANGE REPORT (DCR)

Originator: ____________ Date: ____________

Organization: ____________ Phone No.: ____________

Manual Theoretical ____________

User's ____________ Page

Programmer's ____________ Numbers

Demonstration ____________

Description and reason of change:

(Attach a copy of the page(s) to be changed with corrections typed. Use separate pages if necessary.)

Comments:

Editor Approval	ANOPP Approval	DPSL Entry	Change Made	ANOPO Verif.	Editor Verif.	DPSL Entry
Date	Date	Date	Date	Date	Date	Date

Figure 1. ANOPP DOCUMENTATION CHANGE REPORT (DC)

rejection or acceptance of the change. If the change is unacceptable, the submitter will be so informed and told what action must be taken prior to resubmission. If the change is acceptable, the maintenance organization will be informed so the change can be incorporated into existing mats and magnetic cards. If the change affects software, a Software Change Report (SCR) must be submitted with the DCR. If the change affects more than one manual, those manuals and page numbers must be indicated in the COMMENTS of the DCR.

All changes will be rigidly controlled, reviewed, cataloged, accounted, and filed. Documentation Page Status Logs (DPSL) will be maintained for and in each manual. A Documentation Change Report Status Log will be maintained with the changes for each manual. If a change affects more than one manual, it will be checked on the primary Documentation Change Request Report Status Log by the DCR number.

EXECUTIVE MODULES

3.1 OVERVIEW

Chapter 3 provides a thorough description of the ANOPP system including labeled common blocks, executive control structures, executive data base structures, the Executive Management System (EM), Data Base Management System (DBM), Dynamic Storage Management System (DSM), the Update Utility, and other general utilities.

The Executive Management System controls the execution of ANOPP from beginning to end. The Executive Monitor (XM), described in Section 3.5.3, calls into execution the eight phases of execution described in Section 3.5.4. The course of execution is dependent on the control statements found in the Primary Input Stream. A complete description of these control statements is found in Section 3.5.2.

The Data Base Management System described in Section 3.6 provides ANOPP with a means of storing and retrieving data on sequential and direct access storage devices. DBM provides user callable routines for accessing and manipulating the data base structures described in Section 3.4.

The Dynamic Storage Management System described in Section 3.7 provides ANOPP with a means of getting and freeing various sized blocks of available core storage and making them directly addressable by the requesting module. Many of the core-resident control structures described in Section 3.3 are allocated and manipulated via functions of the DSM.

The General Utilities described in Section 3.9 are a collection of general purpose subprograms available for usage by all executive system routines. Most of the general utility modules are also available for use by functional modules.

The UPDATE control statement is described under the Executive Management System but is explained in more detail in Section 3.8. UPDATE provides a means of building a new data unit either from an existing data unit used as a basis for modification or from one or more data members on one or more data units.

3.2 LABELLED COMMON BLOCKS

The FORTRAN data structure called labelled common is used in ANOPP implementation for reasons of efficiency and security. It is efficient in terms of storage and execution time for several modules that require access to a common set of parameters to share them in common storage rather than continually passing them as arguments in lengthy calling sequences. Security is maintained on a need to know basis by including the labelled common statement in only those modules that share the requirements for availability of the parameters.

All of the labelled common blocks used by the ANOPP executive system are described in the following sections in terms of their primary purpose followed by a list of modules that reference them.

3.2.1 /XBSC/

Common block /XBSC/ contains those variables used during the Initialization Phase by XBS to initialize various executive system tables. For further description of the variables in /XBSC/, see the prologue of Block Data XBSCBD.

Those Executive Modules which use /XBSC/ are:

XBS XBSCBD XFMANT

3.2.2 /XCAC/

Common block /XCAC/ contains variables used during the Secondary Edit Phase by XCA. For further description of the variables in /XCAC/, see the prologue of Block Data XCACBD.

Those Executive Modules which use /XCAC/ are:

XCA XCABD XCABST XCACLØ XCAI XCAMST
XCAMXX XCANCS XCANS XCANWC

3.2.3 /XCRC/

Common block /XCRC/ contains those variables used by XCR, the executive crack module. For further description of the variables in /XCRC/, see the prologue of Block Data XCRCBD.

Those Executive Modules which use /XCRC/ are:

XCR XCRCBD XCRCH XCADØT XCRDR XCREF
XCREXP XCRFC XCRILL XCRPD XCRPH XCRPN
XCRPØT XCRREN XCRSEN XCRSNM XCRWC XCRWCH

3.2.4 /XCS/

Common block /XCS/ contains those variables used in processing or building control statement records. For further description of the variables in /XCS/, see the prologue of Block Data XCSBD.

Those Executive Modules which use /XCS/ are:

XAR XAT XBS XBSDBM XBSDSM XBSBCS
XCSBD XCSIL XCSLØG XCSP XCSPM XCSSL
XCT XDT XEX XEXA XEXL XFM
XFMANT XGØ XIF XLINK XMERR XMERRI
XMERRL XPA XPAVTB XPU XPUTP XRT
XRTAMU XRTBAD XRRBCS XRTBLR XRTCAL XRTCSS
XRTDAT XRTI XRTLRF XRTLSE XRTSER XRTSEX
XRTSIF XRTSSS XRTSYN XRTU XRTVCS XSS
XTB XUN XUPADS XUPCS XUPDIR XUPSRC
XUPSYN

3.2.5 /XCSFM/

Common block /XCSFM/ contains those parameters used by the Executive System Modules, including those which maintain interface functions between the control statement stream and the functional module. For further description of the variables in /XCSFM/, see the prologue of Block Data XCSFMBD.

Those Executive Modules which use /XCSFM/ are:

XASKP XBS XBSSP XBSTP XCSFMBD XCSP
XCSST XEX XEXA XFAN XFMANT XGETP
XIF XPA XPAGE XPAVTB XPLAB XPLABQ
XPLINE XPUTP

3.2.6 /XCSPC/

Common block /XCSPC/ contains those variables during the Control Statement Processing Phase by XCSP. For futher descriptions of the variables in /XCSPC/, see the prologue of Block Data XCSPCBD.

Those Executive Modules which use /XCSPC/ are:

```
XAR      XAT      XCSIL    XCSLØG   XCSP     XCSPBD
XCSPM    XCSSL    XCT      XDR      XDT      XEX
XEXA     XEXL     XGØ      XIF      XMERR    XMERRI
XMERRL   XPA      XPU      XSS      XTB      XUN
XUPCS    XUPLST   XUPNEW   XUPSRC
```

3.2.7 /XCVT/

Common block /XCVT/ contains general variables used by various Executive System Modules. For further description of the variables in /XCVT/, see the prologue of Block Data XCVTBD.

Those Executive Modules which use /XCVT/ are:

```
DSMERR   ILSHFT   IMASK    IKSHFT   ISHIFT   MMBAME
MMBFSI   MMBFST   MMBFT8   MMCLØS   MMCLSE   MMGED
MMPFMT   MMSAND   MMSUD    MMUHMD   MMUPMD   MMVTD
NUMTYP   NWDTYP   TMCLØS   TMEDTB   TMFTE    TMMØPN
TMSTD    TMTØPN   XAR      XASKP    XAT      XBS
XBSDBM   XBSDSM   XBSGCS   XBSIN    XBSSP    XBSTP
XCA      XCABST   XCAT     XCAMST   XCAMXX   XCANCS
XCANS    XCANSP   XCANWC   XCATRA   XCR      XCRCF
XCRCH    XCRDØT   XCRDR    XCRFC    XCRILL   XCRPD
XCRPH    XCRPN    XCRPØT   XCRPS    XCRSRD   XCRWC
XCRWCH   XCSCCS   XCSCIL   XCSP     XCSPM    XCSST
XCT      XCDBDU   XCDBMD   XCTDU    XCUTBD   XDR
XDT      XEX      XEXA     XFAN     XFETCH   XFM
XFMANT   XFMDSM   XFMMM    XFMTM    XGETP    XIF
XM       XMERR    XMERRI   XMPRT    XPA      XPAGE
XPAVTB   XPK      XPKM     XPU      XPUTP    XRE
XRT      XRTBAD   XRTBCS   XRTBLR   XRTCAL   XRTEND
XRTI     XRTLRF   XRTØBD   XRTPIN   XRTSIF   XRTSYN
XRTU     XRTVCS   XSTØRE   XTB      XTBERR   XTRACE
XT1AL    XT1FV    XT2AL    XT3FL    XT3FV    XT3IF
XT3LK    XUN      XUNALL   XUNBGN   XUNCCS   XUNLUH
XUNPK    XUNPKM   XUP      XUPADD   XUPADS   XUPALL
XUPCDT   XUPCGP   XUPCHG   XUPCHI   XUPCHS   XUPCHX
XUPCIN   XUPCØS   XUPCRY   XUPCQD   XUPCQT   XUPCS
XUPDIR   XUPECE   XUPECI   XUPINS   XUPNEW   XUPNMT
XUPØMS   XUPØMT   XUPPRE   XUPSRC   XUPSUM   XUPSYN
XUPXCR   XUPXFR   XVNAME
```

3.2.8 /XDBMC/

Common block /XDBMC/ contains those variables used by the Data Base Management System (DBM). For further description of the variables in /XDBMC/, see the prologue of Block Data XDBMCBD.

The Executive Modules which use /XDBMC/ are:

MMBAME MMBFSI MMBFST MMBFT1 MMBFT8 MMBFT9
MMBMCI MMBMH MMCLØS MMCLSE MMCRMX MMDØMC
MMEDNM MMERR MMFEFB MMGED MMGEFB MMGET
MMGETE MMGETR MMGETW MMGNEW MMGNWE MMIØMC
MMMDMH MMNWR MMØPRD MMØPWD MMØPWS MMPFMT
MMPØSN MMPUT MMPUTE MMPUTR MMPUTW MMREW
MMRMD MMRMH MMRRS MMSAMD MMSFEI MMSKIP
MMSUD MMUHMD MMUPMD MMVBA MMVNM MMVUM
XAR XAT XBSDBM XCT XCTBDU XCTBMD
XCTDU XDBMCBD XDR XDT XFMMM XFMTQ
XPU XUN XUNALL XUNBGN XUNCCS XUNCPY
XUPADD XUPALL XUPCGP XUPCHG XUPNEW XUPXCR
XUPXFR

3.2.9 /XDSMC/

Common block /XDSMC/ contains variables required by the Dynamic Storage Management System (DSM). For further description of the variables in /XDSMC/, see the prologue of Block Data XDSMCBD.

Those Executive Modules which use /XDSMC/ are:

DSMB DSMCØN DSMDFB DSMERR DSMET DSMEUX
DSMF DSMFLB DSMG DSMGUB DSMI DSMIDS
DSML DSMQ DSMR DSMS DSMU DSMX
DSMXFB DSMIST XDSMCBD XFMDSM

3.2.10 /XDTMC/

Common block /XDTMC/ contains variables required by the Table Manager Module. For further description of the variables in /XDTMC/, see the prologue of Block Data XDTMCBD.

Those Executive Modules which use /XDTMC/ are:

TMBLD1 TMCLØS TMEDTB TMERR TMFTE TMGEN1
TMMØPN TMØPN TMØPNA TMSTD TMTABP TMTERP
TMTØPN XBSDBM XDTMCBD XFMTM XTB XTBADV
XTBAIV XTBLD1 XTBPNC XTBVAR

3.2.11 /XPØTH/

Common block /XPØTH/ contains those variables used by the module XCRPØT and its submodules. For further description of the variables in /XPØTH/, see the prologue of Block Data XPØTHBD.

Those Executive Modules which use /XPØTH/ are:

XCRDØT XCRDR XCREXP XCRFC XCRPH XCRPØT
XCRSRD XCRWCH XPØTHBD

3.2.12 /XRØØT/

Common block /XRØØT/ contains those variables used during the Primary Edit Phase by XRT. For further description of the variables in /XRØØT/, see the prologue of Block Data XRØØTBD.

Those Executive Modules which use /XRØØT/ are:

XRØØTBD XRT XRTBAD XRTBCS XRTBLR XRTCAL
XRTCSS XRTDAT XRTEND XRTI XRTLRF XRTLSA
XRTLSE XRTPIN XRTRS XRTSER XRTSEX XRTSIF
XRTSSS XRTSYN XRTTC XRTU

3.2.13 /XSLF/

Common block /XSLF/ contains those variables used in creating and using sequential library files. For further description of the variables in /XSLF/, see the prologue of Block Data XSLFBD.

Those Executive Modules which use /XSLF/ are:

XSLFBD XUN XUNALL XUNBGN XUNCCS XUNCPY
XUNEND XUNLUH

3.2.14 /XSPT/

Common block /XSPT/ contains those Executive System Parameters which may be set by the user via a SETSYS control statement. For further description of the variables in /XSPT/, see the prologue of Block Data XSPTBD.

Those Executive Modules which use /XSPT/ are:

XBS XBSSP XCA XCAMXX XCANCS XCANWC

XCSP XMERR XRT XRTDAT XRTEND XRTPIN

XSPTBD XSS

3.2.15 /XUPC/

Common block /XUPC/ contains those variables used by the UPDATE modules (XUP). For further description of the variables in /XUPC/, see the prologue of Block Data XUPCBD.

Those Executive Modules which use /XUPC/ are:

XUP XUPADD XUPADS XUPALL XUPCDT XUPCGP

XUPCHG XUPCHI XUPCHS XUPCHX XUPCIN XUPCØS

XUPCPY XUPCQD XUPCQT XUPCS XUPDIR XUPINS

XUPLST XUPMLV XUPNEW XUPNMT XUPØMS XUPØMT

XUPØST XUPPRE XUPRLV XUPSRC XUPSUM XUPSYN

XUPXCR XUPXFR

3.3 EXECUTIVE CONTROL STRUCTURES

This section includes a graphical layout and a usage description of all primary control structures used and referenced by executive modules. A control structure is a table, a directory or any other information block which is core resident and not residing on a data unit/member. An information block which is both core resident and data unit/member resident is classified as a data base structure and is included in Section 3.4.

These control structures residing in core are generally addressable in two ways; either as indexed arrays from 1 to n, or as a block of dynamic storage indexed relative to the FORTRAN variable IX in system labelled common block /XANØPP/ plus a positional offset from the start of the block. The dynamic storage index is referred to generically in this manual as the IDX of the block. See the following example which addresses the array TBL from 1 to n or correspondingly the block IX(IDXTBL) plus 0 to n-1.

TBL(1)	IX(IDXTBL+0)
TBL(2)	IX(IDXTBL+1)
TBL(3)	IX(IDXTBL+2)
.	.
.	.
.	.
.	.
.	.
.	.
TBL(n)	IX(IDXTBL+n-1)

The positional offset constants 0 to n-1 have been parameterized by using FORTRAN variables containing constant values to reference offset table entries in many of the ANOPP executive control structures.

3.3.1 System Table Types

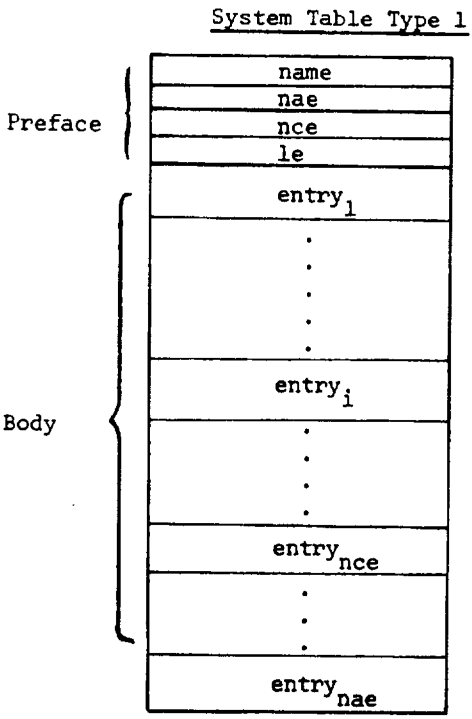
Many of the tables and directories maintained by ANOPP system modules have a common structure. This structure has two parts, a preface and a body. The preface describes the table's current status and the body contains the entries, which may be fixed or variable length depending on the particular table definition. A table which has this common structure is designated as a System Table. Executive Utilities are available for performing various functions for a System Table.

There are three types of System Tables. The structure of the three types are described below.

3.3.1.1 System Table Type 1

Description: The System Table Type 1 structure provides for fixed length table entries. Each entry made in the table requires the same number of words. The positions of words in the preface and the first word beyond the preface have been parameterized by variables in the common block /XCVT/.

Format:



not used currently

Position Parameter	Common Block
NTNAME	/XCVT/
NTMAX	
NTCUR	
NTENT	
NTSTRT	

name	-	name of table (Hollerith)
nae	-	number of allocated entries (integer)
nce	-	number of current entries (integer)
le	-	length of an entry in words (integer)
entry_i	-	an entry of length le

The total length of a type 1 table is the sum of preface length and body length where body length is the product of the number of allocated entries and the length of an entry. For a type 1 table in dynamic storage at IDXT1, the expression for its length, LENT1, would be:

$$\text{LENT1} = \text{NTSTRT} + \text{IX(IDXT1+NTMAX)*IX(IDXT1+NTENT)}$$

3.3.1.2 System Table Type 2

Description: The System Table Type 2 structure provides for variable length table entries. The number of words required for an entry is not necessarily the same as any or all of the other table entries. The user of this table structure must devise his own plan to access the individual entries in the table, if there is more than one entry in the table. The positions in the preface and the first word beyond the preface have been parameterized similar to type 1 tables with the exception of the fixed length of an entry which has no meaning for type 2 tables.

Format:

	System Table Type 2		Position Parameter	Common Block
Preface	name		NTNAME	/XCVT/
	naw		NTMAX	
	ncw		NTCUR	
	not used			
Body	entry_1		NTSTRT	
	. . .			
	entry_i			
	. . . .	naw-ncw		

name - name of table (Hollerith)
naw - number of allocated words in body (integer)
ncw - number of current words in body (integer)
entry_i - an entry of variable length

The total length of a type 2 table is the sum of preface length and body length where body length is the number of allocated words. For a type 2 table in dynamic storage at IDXT2, the expression for its length, LENT2, would be:

LENT2 = NTSTRT + IX(IDXT2+NTMAX)

3.3.1.3 System Table Type 3

Description: The System Table Type 3 is characterized by forward chained, fixed length entries. These entries are linked into one of three chains -- the used entry chain, the free entry chain, or the other entry chain. Each entry contains a chain control word, which serves as a forward pointer to its successor in the chain, followed by space reserved for the user's entry data. A chain control word whose value is zero indicates the end of the chain. Each position in the preface plus the first word of the first entry have been parameterized by variables in common block /XCVT/.

Free entries may exist anywhere in the body of the table, not necessarily the last entry. The table can accommodate up to three chains.

Format:

	System Table Type 3	Position Parameter	Common Block
Preface	name	NTNAME	/XCVT/
	nae	NTMAX	
	ctl	NTCUR	
	le	NTENT	
	uecp	NT3USD	
	fecp	NT3FRE	
	oecp	NT3ØTR	
Body	entry_1	NT3STR	
	⋮		
	entry_i		
	⋮		
	entry_{nae}		

name - name of table (Hollerith)
nae - number of allocated entries (integer)
ctl - current table length in words (integer)
le - length of an entry (integer)
uecp - used entry chain pointer index to the first entry in the used entry chain
fecp - free entry chain pointer index to the first entry in the free entry chain
oecp - other entry chain pointer index to the first entry in the other entry chain
$entry_i$ - entry of length le including the chain control word

The total length of a type 3 table is the sum of preface length and body length where body length is the product of the number of allocated entries and the length of an entry. For a type 1 table in dynamic storage at IDXT3, the expression for its length, LENT3, would be:

```
LENT3  =  NT3STR + IX(IDXT3+NTMAX)*IX(IDXT3+NTENT)
```

3.3.2 Active Member Directory (AMD)

System Table Type: 3

Residence: Global Dynamic core; the IDX is IDXAMD in /XDBMC/ common block.

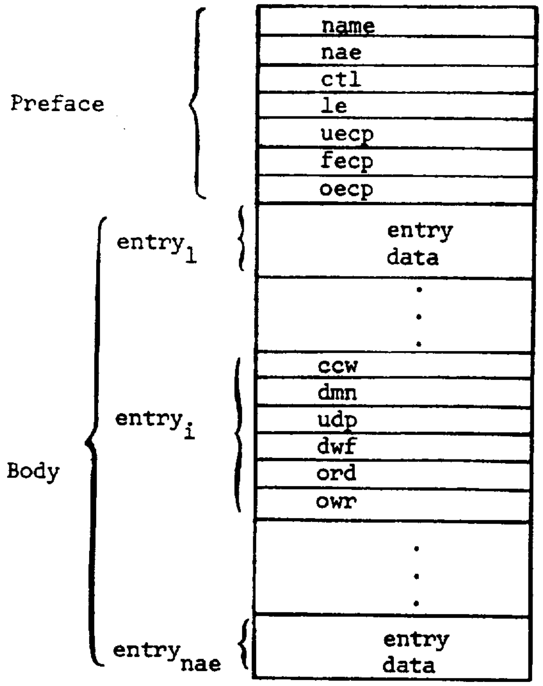
Primary Users: Data Member Manager and Data Table Manager open and close routines (MMØPRD, MMØPWD, MMØPWS, MMCLØS, TMØPN, and TMØPNA) and XFMMM which logically closes active members that remained open following termination of a functional module.

Description: The AMD is a table identifying all data members which are open to Data Member Manager (MM) and provides a linkage to the NAME argument used to open a data member and to the Data Unit Directory entry for the data unit named in the open member request. Additionally, an AMD entry indicates whether a data member is open for input, output, or both, and if open for output, whether it is open for direct or indirect writing.

The AMD is allocated during ANOPP initialization and remains resident throughout an ANOPP run. Expansion of the AMD occurs as required.

Format:

Active Member Directory



Position Parameter	Common Block
(see system table type 3 preface)	
IAMDMN	/XDBMC/
IAMUDP	
IAMDWF	
IAMØRD	
IAMØWR	

ccw - chain control word linking the entry into the free or used chain Other chain is invalid for the AMD.
dmn - data member name (Hollerith)
udp - Data Unit Directory entry pointer which is an index relative to the beginning of the DUD
dwf - direct write flag
ord - open read control word which contains the IDX of the NAME array used if the data member was opened for reading via MMØPRD
owr - open write control word which contains the IDX of the NAME array used if the data member was opened for writing via MMØPWD or MMØPWS

<u>Initialization</u>: During ANOPP initialization XBSDBM creates the AMD using the following variables:

1. The number of words of dynamic core initially allocated is determined using the following formula:

$$\text{LEN} = \text{NT3STR} + \text{NAEAMD*LENAME}$$

NT3STR is a variable from /XCVT/ common block, and NAEAMD and LENAME are from /XDBMC/ common block.

2. The AMD table preface is initialized as follows:

name	=	IDAMD from /XDBMC/ common block
nae	=	NAEAMD from /XDBMC/ common block
ctl	=	LEN which was computed in 1. above
le	=	LENAME from /XDBMC/ common block
uecp	=	zero
fecp	=	NT3STR+1
oecp	=	zero

3. The body of the AMD is initialized in system table type 3 format using subprogram XT3IF.

<u>Entry</u>: An entry is made in the AMD each time a previously unopened data member is opened via MMØPRD, MMØPWD, or MMØPWS.

<u>Retrieval</u>: The AMD used entry chain is searched each time a request to open a data member occurs. If a matching entry is found and it is not open for the mode specified by the open request, the entry is updated according to the mode (read or write) of the open request.

The AMD is also searched by Data Table Manager when a data table is being opened to prevent a data member from being open to both Data Member Manager and Data Table Manager.

Deletion: When a data member is closed for both input and output processing modes, its AMD entry is cleared and linked into the free entry chain.

3.3.3 Alternate Names Table (ANT)

System Table Type: 1

Residence: Global dynamic core; the IDX is LANT in /XCSFM/ common block.

Primary Users: Data Member Manager, Data Table Manager, and the Parameter Maintenance Functions (XASKP, XPUTP, XGETP).

Description: The ANT is a table of reference names and corresponding alternate names as specified on the EXECUTE CS. Alternate names exist only during the F.M. Processing Phase when the specified F.M. is executed; at other times, the ANT is a null table.

Format:

		Alternate Names Table	Position Parameter	Common Block
Preface		name	(see System Table Type 1 Preface)	
		nae		
		nce		
		le		
Body	entry_1	refname		
		altname		
		. . . . .		
	$\text{entry}_{\text{nce}}$	$\text{refname}_{\text{nce}}$	LANTN	/XCSFM/
		$\text{altname}_{\text{nce}}$	LANTA	

refname - reference name specified on EXECUTE CS (Hollerith)
altname - corresponding alternate name specified on EXECUTE CS (Hollerith)

Initialization: ANT is allocated for zero entries during the Initialization Phase (XBS) and is reinitialized for zero entries upon completion of the Functional Module Processing Phase.

1. The number of words of dynamic core initially allocated is determined by:

$$\text{LEN} = \text{NTSTRT}$$

NTSTRT is a variable from /XCVT/ common block.

2. The ANT preface is initialized as follows:

```
name = NAMANT from /XBSC/ common block
nae  = NAEANT from /XBSC/ common block
nce  = NCEANT from /XBSC/ common block
le   = LEANT from /XBSC/ common block
```

Entry: When an EXECUTE control statement is processed during Control Statement Processing Phase, the ANT is allocated in GDS for exact number of entries required. An entry for each reference/alternate name specified is made and the values for nae and nce are updated.

Retrieval: Utility XFAN (fetch alternate name) provides retrieval. MM and TM user calls and the utilities XPUTP, XASKP, and XGETP retrieve alternate names automatically on each call.

Deletion: All entries deleted upon completion of the functional module specified on the EXECUTE control statement by the XFMANT module.

3.3.4 Data Table Directory (DTD)

System Table Type: 3

Residence: Global dynamic core; the IDX is IDXTD in /XDTMC/ common block.

Primary Users: Data Member Manager and Data Table Manager open and close modules (MMØPRD, MMØPWD, MMØPWS, TMØPN, TMØPNA, TMCLØS) and XFMTM which logically closes data tables left open following termination of a functional module.

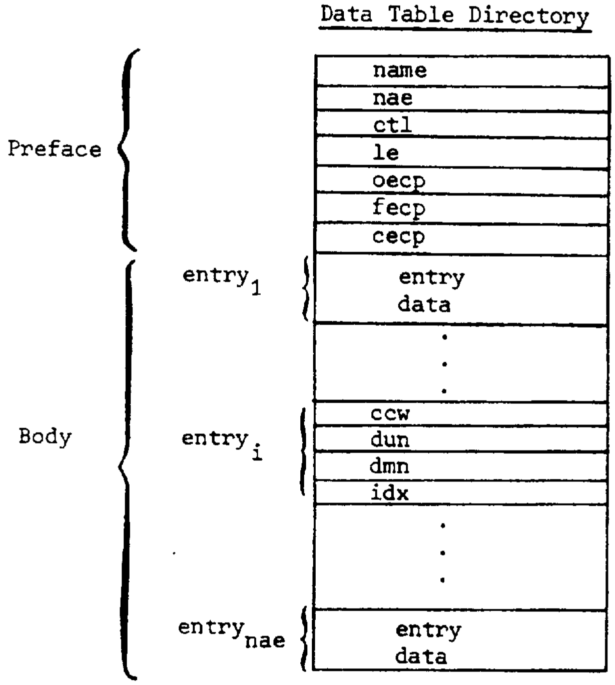
Description: The DTD identifies all data tables, both open and closed, which are core resident at any point in time during an ANOPP run. A DTD entry contains a data unit and data member name, which uniquely identify a data table, and an IDX variable which is used in the following two ways:

1. When a data table is open, the IDX variable contains the IDX to the NAME argument used in opening the table and the third word of the NAME argument contains the IDX to the data table.

2. When a data table is closed, the IDX variable contains the IDX to the data table.

The DTD is allocated during ANOPP initialization by XBSDBM and remains resident until the ANOPP run completes. The DTD cannot be expanded dynamically and therefore ANOPP will terminate abnormally if the user tries to simultaneously open more data tables than there are entries in the DTD.

Format:



Position Parameter	Common Block
(See System Table Type 3 Preface)	
ITEDUN ITEDMN ITEIDX	/XDTMC/

oecp - open entry chain pointer index to the first entry in the open entry chain
fecp - free entry chain pointer index to the first entry in the free entry chain
cecp - closed entry chain pointer index to the first entry in the closed entry chain

There are three entry chains in the body of the DTD; the "open" entry chain, the "free" entry chain, and the "closed" entry chain. Entries in the open entry chain contain the following entry data:

ccw - chain control word linking the entry into the open chain
dun - name of the data unit on which the data table resides
dmn - name of the data member which contains the data table
idx - index, relative to /XANØPP/ common block, of the NAME argument used in opening the data table

Entries in the free entry chain have only a chain control word and the entry data is zero.

Entries in the closed entry chain contain the following entry data:

ccw - chain control word linking the entry into the closed entry chain
dun - as described previously
dmn - as described previously
idx - index, relative to /XANØPP/ common block, of the data table which is located in global dynamic core

Initialization: During ANOPP initialization, XBSDBM creates the DTD using the following:

1. The number of words of dynamic core initially allocated is determined using the following formula:

   LEN = NT3STR + NAETD * LENTDE, where

   NT3STR is a varible from /XCVT/ common block, and NAETD and LENTDE are from /XDTMC/ common block.

2. The DTD preface is initialized as follows:

   name = from IDTD in /XDTMC/ common block
   nae = from NAETD in /XDTMC/ common block
   ctl = from LEN computed in 1. above
   le = from LENTDE in /XDTMC/ common block
   oecp = 0
   fecp = 1 + NT3STR
   cecp = 0

Entry: A new entry is made in the DTD when a data table which is not currently in the open entry chain or the closed entry chain is opened.

Retrieval: Entries are retrieved from both the open and closed entry chains by Data Table Manager (DTM) open and close modules. The DTM open data table modules (TMØPN, TMØPNA) link DTD entries from the closed entry chain into the open entry chain and the close data table module (TMCLØS) links from the open to the closed entry chain.

Deletion: Data Member Manager (DMM) open data member modules (MMØPRD, MMØPWD, and MMØPWS) also search the open and closed entry chains in the DTD for a data table residing on a particular data unit and member. However, if DMM finds an entry, either the DMM module involved abnormally terminates ANOPP if the entry is in the open entry chain or it frees the data table from core and links the DTD entry from the closed to the free entry chain.

3.3.5 Data Unit Directory (DUD)

System Table Type: 3

Residence: Global dynamic core; the IDX is IDXUD in /XDBMC/ common block.

Primary Users: All DBM control statements, the UPDATE control statement, all Data Member Manager modules, and Data Table Manager open data table modules.

Description: The DUD identifies all data units which are available to ANOPP at any one time during ANOPP execution. A DUD entry contains a copy of the data unit header for the unit, an IDX linking the entry to the external file information table and buffer, and other identification and control information.

The DUD is allocated during ANOPP initialization by subprogram XBSDBM and remains resident until ANOPP termination. The DUD must be one of the control structures allocated at the beginning of global dynamic core and may not be expanded. This insures that the DUD does not move during DSM consolidation of global dynamic core.

Format:

		Data Unit Directory		Position Parameter	Common Block
Preface		name		(See System Table Type 3 Preface)	
		nae			
		ctl			
		le			
		uecp			
		fecp			
		oecp			
Body	entry_1	entry data			
		. . .			
	entry_i	ccw	Data Chart Header		
		id		IUHID	/XDBMC/
		af		IUHAF	
		nwa		IUHNWA	
		wa		IUHMDA	
		len		IUHMDL	
		dun	Data Unit Control Info.	IDUDUP	
		efn		IDUEFN	
		cbi		IDUCBI	
		pbi		IDUPBI	
		omc		IDUOMC	
		dwf		IDUDWF	
		. . .			
	$\text{entry}_{\text{nae}}$	entry data			

entry data:

ccw - chain control word linking the entry into the free or used chain. The other chain is invalid in DUD.

id - Data Unit Header identifier (Hollerith)

af - integer ARCHIVE flag, if equal to zero, write is permitted on the data unit. If equal to 1, then writing is not permitted.

nwa - integer next word address that is available for writing on the data unit

wa - integer word address of the Data Member Directory on the data unit

len - integer length (in words) of the Data Member Directory

dun - data unit names used for the data unit in the current ANOPP run

efn - external file name assigned to the data unit by the operating system

cbi - the IDX to the external file information and buffer in global dynamic core

pbi - the "previous buffer index" contains the value of cbi from the previous I/O operation. It is used to determine if the IDX to the buffer has been changed since the last I/O operation

omc - integer open member count; indicating the number of data members currently open on the data unit dwf - integer direct write flag which indicates that a data member is open to write directly on the unit

Initialization: During ANOPP initialization, subprogram XBSDBM creates the DUD using the following:

1. The number of words of dynamic core initially allocated is determined using the formula:

   LEN = NT3STR + NAEUD * LENUDE, where

   NT3STR is a variable from /XCVT/ common block, and NAEUD and LENUDE are from /XBDMC/ common block.

2. The DUD preface is initialized as follows:

   name = from IDUD in /XDBMC/ common block
   nae = from NAEUD in /XDBMC/ common block
   ctl = from LEN computed in 1. above
   le = from LENUDE in /XDBMC/ common block
   uecp = 0
   fecp = NT3STR + 1
   oecp = 0

3. The body of the DUD is initialized using subprogram XT3IF which builds the free entry chain.

Entry: A new entry is made in the DUD whenever a CREATE, ATTACH, or LØAD control statement is processed. Also, use of Data Member Manager's (DMM) open for indirect writing facility causes creation of a temporary entry in the DUD for each data member opened.

Retrieval: The DUD is searched each time a DBM control statement, an UPDATE or a Data Table Manager open or close request is processed. Also, when a data member is open indexes to the related DUD entry (or entries if open for indirect write) are retained in the member's AMD entry and MCB. These indexes are used to directly access the DUD entry for all DMM input and output processing.

Deletion: Entries are deleted from the DUD by the DETACH and PURGE control statements, and, when a data member which was open for indirect writing is closed, its temporary (scratch) data unit is purged and the DUD entry is deleted.

3.3.6 Member Control Block (MCB)

System Table Type: Not applicable

Residence: Global dynamic core; the IDX is the third word of the NAME argument used in opening the data member.

Primary Users: Data Member Manager Modules.

Description: The MCB is the primary control structure used in building and accessing data members. It provides indexes to the Data Unit Directory and Active Memory Directory entries which relate to the data member and control information regarding current record being read or written and position within record. In addition, it contains the Data Member Header, Record Directory, and one Record Subdirectory.

The MCB is allocated when a data member is opened and is resident until the member is closed. Reallocation of the MCB takes place only when opening a data member for reading. Expansion is required then to provide space for a Record Subdirectory.

Format:

Member Control Information	id
	len
	indud
	inamd
	inrd
	inrs
	crn
	rwc
	infst
	wadmh
Data Member Header	(see Data Base Structures)
Record Subdirectory	(see Data Base Structures)

The MCB consists of three separate structures which are necessary to control member data input and output:

1. Member Control Information (MCI)
2. Data Member Header (DMH), and
3. Record Subdirectory (RS).

Since the DMH and RS are discussed in the Data Base Structures section, only the MCI is described here.

MCI:

id - MCB identifier (Hollerith)

len - integer length (in words) of the MCB

indud - index, relative to the beginning of the DUD, to the DUD entry describing the data unit to which the data member belongs

inamd - index, relative to the beginning of the AMD, to the AMD entry for the data member described by the MCB

inrd - index, relative to the beginning of the Master Record Directory (RD), to the RD entry for the Record Subdirectory (RS) current in the MCB

inrs - index, relative to the beginning of the RS, to the RS entry for the current data record

crn - integer current record number; MMSKIP and MMPØSN modify this field to provide random accessing of data

rwc - record word count; this field is used to determine the current position within a record for partial record gets and puts. It contains the count of the number of words transferred to or from a record

infst - FST index is used for element gets and puts (MMGETE and MMPUTE) to retrieve element type and length from the Format Specification Table (FST) in the DMH

wadmh - integer word address of the DMH, provides MMCLØS with the address at which the DMH is to be written if the member is open to write, or zero if its open to read

Initialization: Not applicable

Entry: Not applicable

Retrieval: The MCB is accessed and modified during every DMM operation.

Deletion: Not applicable

3.3.7 Member Description Blocks Table (MDBT)

System Table Type: 3

Residence: Global dynamic storage; the IDX is MXMDB in /XCS/ common block.

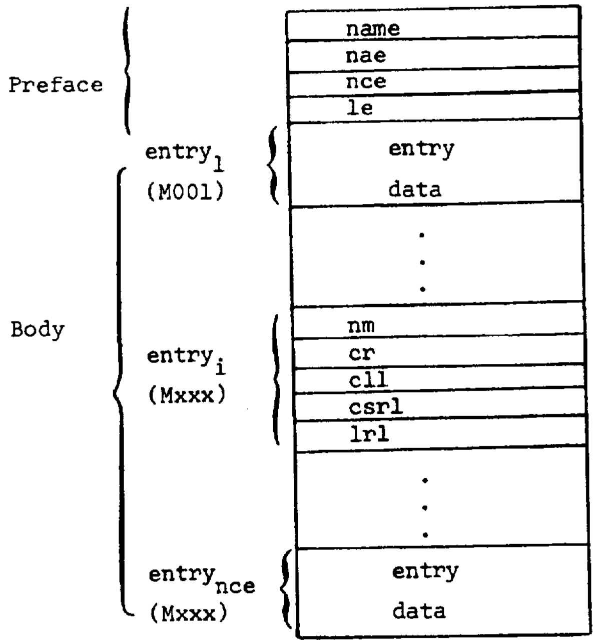
Primary Users: XRT module (Primary Edit Phase) to initialize an entry for an Mxxx name assigned corresponding to a CALL control statement. XRT also constructs the M001 member and puts the MDB in executable format.

XCA module (Secondary Edit Phase) to initialize entries as Mxxx names are assigned and to construct the Mxxx for the CALL being executed and put the corresponding MDB in executable format.

Description: Contains a Member Description Block (MDB) entry for each Mxxx type data member for which an Mxxx name has been assigned. Each entry contains pertinent information about the member, such as member name, number of current CS record in execution, Mxxx that called this Mxxx, maximum length of a CS record, length of label record. The MDB settings indicate if the Mxxx member has been constructed and exists on XSUNIT.

Format:

Member Description Blocks Table



Member Description Blocks Table	Position Parameter	Common Block
name nae nce le	(See system table type 1 Preface)	
nm	MNAME	/XCS/
cr	MCUR	
cll	MCALL	
csrl	MRL	
lrl	MLL	

entry data:

First entry:

nm - M001 - name of root member (Hollerith)
cr - number of current CS record in execution
cll - entry not applicable to M001
csrl - maximum length of a CS record for M001
lrl - number of words in label record for M001

Subsequent entries:

nm - name of Mxxx data member (Hollerith)
cr - number of current CS record in execution
cll - name of Mxxx data member that called this Mxxx member in current execution
csrl - maximum length of a CS record for Mxxx
lrl - number of words in label record for Mxxx

Initialization: The MDBT is allocated and the MDB entry for the M001 member is initialized by XBS.

1. The number of words of dynamic core initially allocated is determined using the formula:

   NWDS = LPREF + NAEMDB * LEMDB, where

   LPREF, NAEMDB, LEMDB are in /XBSC/ common block.

2. The MDBT is initialized as follows:

   name = NAMMDB from /XBSC/ common block
   nae = NAEMDB from /XBSC/ common block
   nce = NCEMDB from /XBSC/ common block
   le = LEMDB from /XBSC/ common block

   entry_1:

   nm = NM001 from /XBSC/ common block
   cr = 0
   cll = blank
   csrl = 0
   lrl = 0

Entry: The M001 MDB entry is put into executable format during the Primary Edit Phase by XRT. The csrl and lrl values are entered for M001 by XRT when M001 is built. Member Description Block entries for other Mxxx members are added to the MDBT as Mxxx member names are assigned during the Primary and Secondary Edit Phase whenever a CALL control statement is edited. When the MDB is added, nm is defined as the Mxxx name assigned, cll the Mxxx calling member, and all other entries are set to zero (entry put into initialized format). The csrl and lrl values will be entered in the MDB (entry put into executable format) the first time the CALL control statement is executed since the Mxxx member is built on the first execution.

Retrieval: Upon entry to the CS Processing Phase (XCSP), the maximum CS record length and label record length are retrieved from the MDB for the Mxxx member in current execution. These lengths are used to allocate LDS blocks for storing CS records and the label record for the Mxxx in current execution. XCSP also retrieves the number of the current CS record in execution from the MDB for the Mxxx in current execution, and uses that value in positioning to the next CS record to be executed. Upon completion of execution of an Mxxx member, XRE processes the RETURN CS and redefines the Mxxx member in current execution as the calling member name found in MCALL of the MDB for the current Mxxx.

Deletion: Once an MDB entry is made in the MDBT it is never deleted.

3.3.8 Sequential Library File Directory (LFD)

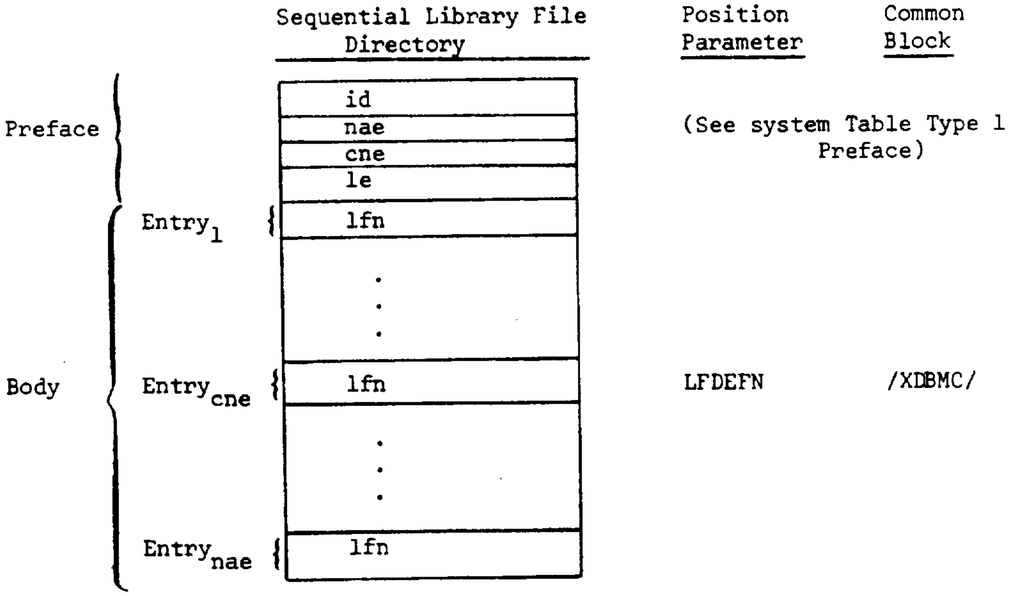
System Table Type: 1

Residence: Global dynamic core; the IDX is IDXLFD in /XDBMC/ common block.

Primary Users: UNLØAD (XUN), LØAD (XLD), and DRØP (XDR) control statements.

Description: The LFD is a table of sequential library file names that is used by the UNLØAD, LØAD, and DRØP control statements to insure the integrity of sequential libraries created or used in a particular ANOPP run. Initially allocated during system initialization, the LFD is resident throughout an ANOPP run and is expanded whenever all allocated entries are in use and additional entries are required. Entries in the LLT are sorted in ascending binary sequence by data member name within data unit name.

Format:



entry data:

lfn - sequential library file name

Initialization: During ANOPP initialization, XBSDBM creates the LFD using the following variables:

1. The number of words of dynamic core initially allocated is determined using the formula:

   LEN = NTSTRT + NAELFD * LENLFE, where

   NTSTRT is a variable from /XCVT/ common block, and NAELFD and LENLFE are from /XDBMC/ common block.

2. The LFD table preface is initialized as follows:

   id = IDLFD from /XDBMC/ common block
   nae = NAELFD from /XDBMC/ common block
   cne = zero
   le = LENLFE from /XDBMC/ common block

3. The body of the LFD is set to zero.

Entry: A new LFD entry results when a unique library file name is encountered in processing a LØAD or UNLØAD control statement.

Retrieval: Entries are sequentially retrieved from the LFD by UNLØAD, LØAD, and DRØP to establish the existence or non-existence of a library file name and their respective control statements.

Deletion: Entries are deleted from the LFD by the DRØP control statement.

3.3.9 Sequential Library Load Table (LLT)

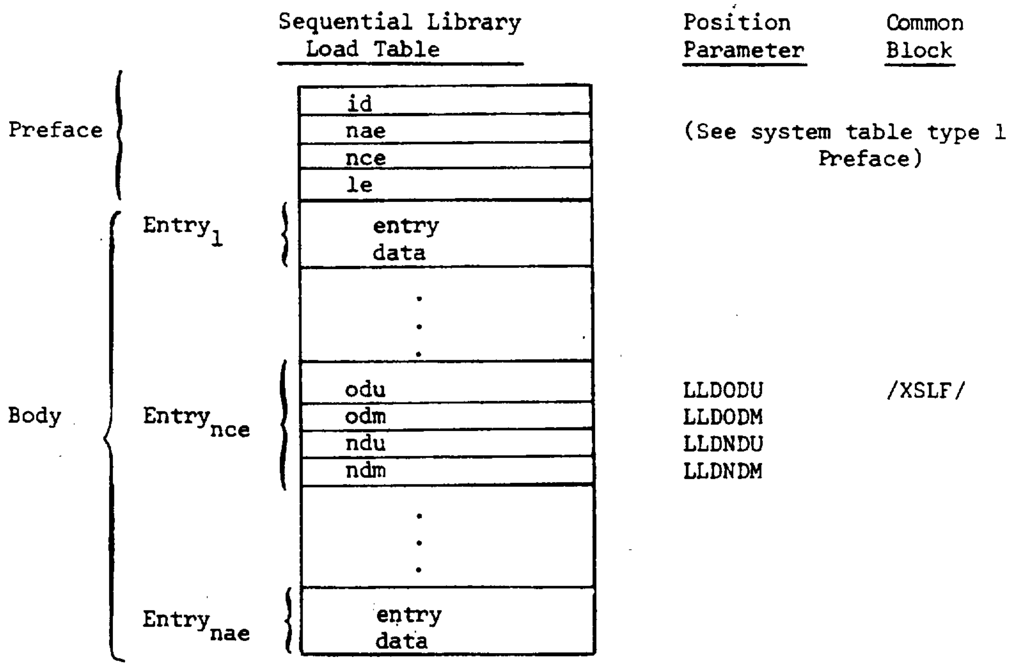
System Table Type: 1

Residence: Local dynamic core; the IDX is IDXLLT in /XSLF/ common block.

Primary Users: LØAD (XLD) control statement.

Description: The LLT is a table of data unit and data member names which is used to control loading and renaming of data unit and members from a sequential library file. Since the LLT is local dynamic core resident, its life span is limited to each period of XLD execution. Expansion occurs at the rate defined by NEXPND in /XCVT/ common block when additional table entries are required.

Format



entry data:

odu - old data unit name
odm - old data member name
ndu - new data unit name
ndm - new data member name

Initialization: The LLT is defined at the beginning of XLD execution by subprogram XLDBGN using the following variables:

1. The number of words of dynamic core initially allocated is determined using the formula:

   $$LEN = NTSTRT + NAELLT * LENLTE$$

   NTSTRT is defined in /XCVT/ common block, and NAELLT and LENLTE are defined in /XSLF/ common block.

2. The LLT table preface is initialized as follows:

   id = IDLLT from /XSLF/ common block
   nae = NAELLT from /XSLF/ common block
   nce = zero
   le = LENLTE from /XSLF/ common block

3. The body of the LLT is initially set to zero.

Entry: An entry is made in the LLT for each data unit and data member named on the LØAD control statement, or, if none were named, for each data unit name in the Library Directory Record in the sequential library file (see Subsection 3.4.5.2). If a data unit or member is renamed on a LØAD control statement then its new name is entered along with the old, otherwise the old and new names will be the same.

Retrieval: Entries are retrieved serially from the LLT as the data units to which they refer are identified and loaded from a sequential library file by subprogram XLD. The new data unit and member names will be used to create the data member.

Deletion: Entries are not deleted from the LLT.

3.3.10 Sequential Library Unit Table (LUT)

System Table Type: 1

Residence: Local dynamic core; the IDX is IDXLUT in /XSLF/ common block.

Primary Users: LØAD (XLD) control statement.

Description: The LUT is a table of data unit names with their related external file names (EFN). It is used to validate their uniqueness against the Data Unit Directory (UD), (see Subsection 3.3.4) and to create UD entries for the data units being loaded.

Allocation of the LUT is done by XLDBGN at the beginning of XLD execution and expansion will occur if the number of unique data units being loaded exceeds the number of allocated entries. Prior to termination, XLD frees the LUT from local dynamic core.

Format:

		Sequential Library Unit Table	Position Parameter	Common Block
Preface		id	(see system table type 1 Preface)	
		nae		
		nce		
		le		
Body	Entry_1	entry data		
		. . .		
	Entry_{nce}	dun	LUTDUN	/XSLF/
		efn	LUTEFN	
		. . .		
	Entry_{nae}	entry data		

entry data:

dun - data unit name
efn - external file name

Initialization: During XLD initialization, XLDBGN creates the LUT in local dynamic core using the following variables:

1. The number of initially allocated words of dynamic core is determined using the formula:

   LEN = NTSTRT + NAELUT * LENUTE, where

   NTSTRT is defined in /XCVT/ common block, and NAELUT and LENUTE are defined in /XSLF/ common block.

2. The LUT table preface is initialized as follows:

   id = IDLUT from /XSLF/ common block
   nae = NAELUT from /XSLF/ common block
   nce = zero
   le = LENUTE from /XSLF/ common block

3. The body of the LUT is initially set to zero.

Entry: An entry is made in the LUT for each new data unit name encountered on the LØAD control statement, or, if all data units are to be loaded, the names of all data units defined in the Library Directory Record (see Subsection 3.4.5.2) from the sequential library file.

Retrieval: Entries are retrieved serially from the LUT and are used by XLDCDU to create Data Unit Directory entries (see Subsection 3.3.4) prior to the loading of data members.

Deletion: Entries are not deleted from the LUT, but the LUT itself is removed from core when processing is complete.

3.3.11 User Parameter Table (UPT)

System Table Type: 1

Residence: Global dynamic core; the IDX is LUPT in /XCSFM/ common block.

Primary Users: EM modules XPA (entry), XPA, XIF (retrieval). Utilities XPUTP (entry), XGETP, XASKP (retrieval).

Description: The UPT is a table of user parameters established in the control statement stream by the PARAM control statement or in a functional module by the Parameter Maintenance Function XPUTP. The parameter values are numerical, logical, or character string values which are maintained in the User Parameter Table or the User String Table.

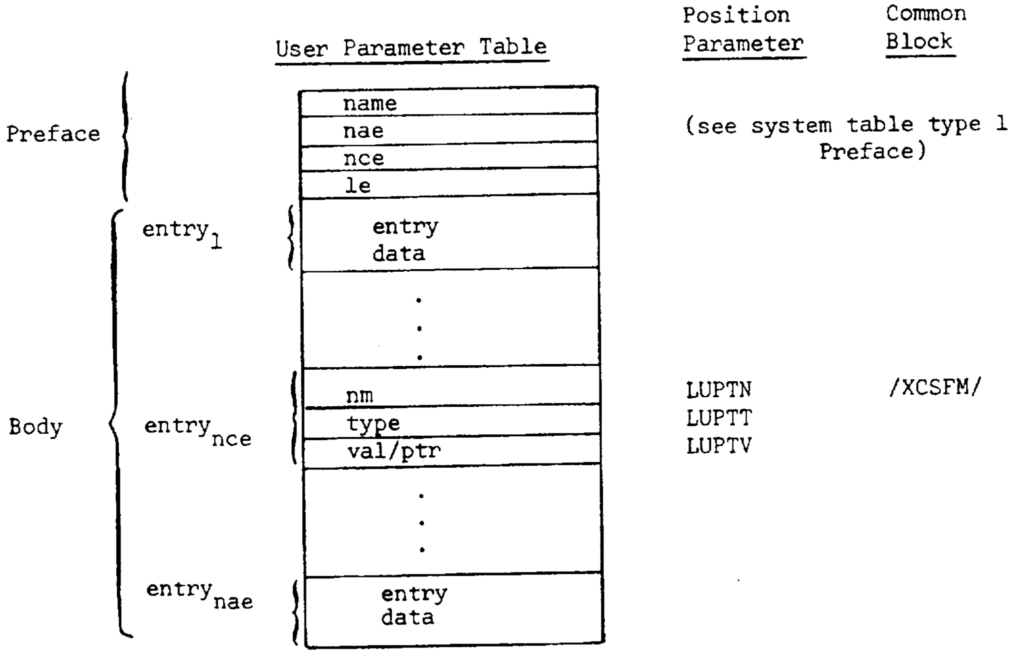
Valid UPT Table Entries:

Type Code	Type Value	Length (words)
1	Integer	1
2	Real Single Precision	1
3	Real Double Precision	2
6	Logical	1
-n	string of n char (A8)	(n+7)/8

The fixed length table entry of four words is provided as the maximum length required for all implemented types except character strings with more than 16 characters. These character string values having more than 16 characters are maintained in the User String Table, and the UPT value entry points to the UST entry. Complex and complex double values have not been provided for in the UPT.

Once a user parameter has been established in the UPT, it may be subsequently retrieved or changed in the control statement stream or in a functional module. The table entries remain throughout ANOPP. Once established, an entry is never deleted from the set of known parameters. The user parameters provide a link in the communication between the control statement stream and a functional module.

Format:



entry data:

nm - name of user parameter
type - integer type code (valid types are 1, 2, 3, 6, -n)
val - if (type .GT.0) or (type=-n and n.LE.16) then value is located in one or two words as required
ptr - if type = -n and n.GT.16 pointer to position in UST of the string relative to start of UST.

Initialization: During the ANOPP Initialization Phase (XBM) the UPT is created using the following:

1. The number of words of dynamic core initially allocated is determined using the formula:

   NWDS = LPREF + NAEUPT * LEUPT, where

   LPREF, NAEUPT, and LEUPT are in /XBSC/ common block.

2. The UPT is initialized as follows:

   name = NAMUPT from /XBSC/ common block
   nae = NAEUPT from /XBSC/ common block
   nce = NCEUPT from /XBSC/ common block
   le = LEUPT from /XBSC/ common block

Entry: Entry is made in the CS Processing Phase by PARAM CS or in the Functional Module Processing Phase by XPUTP. If table becomes insufficient for further entries, GDS block size can be expanded via DSMX by a factor of NEXPND (/XCVT/ common block).

Retrieval: Retrieval from UPT accomplished in the CS Processing Phase by the PARAM and IF control statements or in the Functional Module Processing Phase by the Parameter Maintenance Functions XGETP and XASKP.

Deletion: Once established in the UPT, a parameter is never deleted from the set of known user parameters. There is, therefore, no need for consolidation or reuse of free space.

3.3.12 User String Table (UST)

System Table Type: 2

Residence: Global dynamic core; the IDX is LUST in /XCSFM/ common block.

Primary Users: EM modules XPA (entry), XPA, XIF (retrieval). Utilities XPUTP (entry), XGETP (retrieval).

Description: The UST is a table of the user parameter values which are character strings having more than 16 characters. Entries to this table are made through the control statement stream by the PARAM control statement or in a functional module by the Parameter Maintenance Function XPUTP. A UST entry is associated with an entry in the User Parameter Table (UPT) which names the user parameter, gives its type code (-n) and points to the start of the value entry in the UST.

The number of words required in the UST for the character string is implied by the integer type code in the corresponding UPT entry. Once an entry has been made in the UST, it subsequently may be retrieved or changed in the control statement stream or in a functional module. If the current character string value is being changed and the new value is (a) a type other than character string, (b) a character string with 16 or fewer characters, or (c) a character string requiring more words than the current value, then the current entry in the UST is "delinked" as the pointer in the UPT is changed or overwritten with a value. There is no reuse of the "delinked" character string value or its space in the table. A new entry into the UST always begins at the next available word. Assuming that NC is the number of characters in the character string, then ABS(NC)/NCPW = Q + R (NCPW = number of characters per word). The Q words are copied to the UST. Word Q + 1 contains the R characters, left justified, blank-filled.

Format:

		User String Table	Position Parameter	Common Block
Preface		name	(See system table type 2 Preface)	
		naw		
		ncw		
		(not used)		
Body	entry_1	characters		
		. . .		
	entry_{ncw}	characters		
		. . .		
	entry_{naw}	characters		

entry data:

characters - character string (A8)
number of words is implied by the integer type code in the corresponding UPT entry.

Initialization: During the ANOPP Initialization Phase (XBM) the UPT is created using the following:

1. The number of words of dynamic core initially allocated is determined using the formula:

   NWDS = NPREF + NAWUST, where

   LPREF and NAWUST are in /XBSC/ common block.

2. The UST is initialized as follows:

   name = NAMUST from /XBSC/ common block
   naw = NAWUST from /XBSC/ common block
   ncw = NCWUST from /XBSC/ common block

Entry: Entry is made in the CS Processing Phase by the PARAM control statement or in the Functional Module Processing Phase by XPUTP. If the table becomes insufficient for further entries, GDS block size can be expanded via DSMX by a factor of NEXPND (/XCVT/ common block).

Retrieval: Retrieval from UST is accomplished in the CS Processing Phase by the PARAM and IF control statements or in the Functional Module Processing Phase by the Parameter Maintenance Function XGETP.

Deletion: Deletion from the UST is a result of the following situation. The current length of the user parameter character string is n where n is greater than 16, and the value is to be changed to one of the following:

1. character string with 16 or fewer characters
2. a type other than character string
3. a character string with more than 16 characters which will not fit in the currently allocated UST entry.

The current entry in the UST is "delinked" by the pointer in the UPT entry being changed or overwritten with a value.

3.4 EXECUTIVE DATA BASE STRUCTURES

This section includes a graphical layout and a usage description of all data base structures used and referenced by executive modules.

A data base structure is a table, a directory, or any other information block which resides on a data unit or external file. The general organizational structure of a data unit and a data member are also included as data base structures.

3.4.1 Data Unit

A Data Unit is the highest level of the ANOPP DBM data structure that can be referenced directly using ANOPP control statements. It is physically stored on direct access storage devices and is uniquely identified within an ANOPP run by a data unit name.

Since a data unit resides on a file which is identifiable outside the ANOPP system, its data unit name may be changed from one ANOPP run to another by relating a new data unit name to the same external file name.

3.4.1.1 Data Unit Structure

Residence: Random Access Secondary Storage Devices

Primary Users: Data Base Manager (DBM)

Description: A data unit is a set of data members which is assigned via DBM to an external file defined by the host computer operating system. It contains a Data Unit Header (DUH) and Data Member Directory (DMD) which contain the information necessary to access and add data members.

Format:

Data Unit Structure

DUH
DMD
optional data members

Initialization: When initially created, a data unit contains only the DUH and DMD.

3.4.1.2 Data Unit Header (DUH)

Residence: Data Units

Primary Users: Data Base Manager (DBM) Subprograms

Description: The DUH serves as both a control structure and a data base structure. Although it primarily resides on data units, during ANOPP execution a copy is retained in the Data Unit Directory entry for each data unit.

The DUH contains the information required to access the Data Member Directory (and thereby all data members) and the address of the next word that is available for output. Also, the DUH contains the Archive Flag which can logically inhibit outputting to a data unit.

Format:

Data Unit Header	Position Parameter	Common Block
id	IUHID	/XDBMC/
arflg	IUHAF	
nwa	IUHNWA	
mda	IUHMDA	
mdl	IUHMDL	

id	-	data unit header identifier
arflg	-	unit archive flag
nwa	-	next write address
mda	-	data member directory address
mdl	-	data member directory length

Initialization: When first generated via execution of a CREATE control statement or a call to XCTDU, the DUH has the following values:

id = IDUH from /XDBMC/ common block
arflg = 0
nwa = LENUH+1 where LENUH is from /XDBMC/ common block
mda = nwa
mdl = 0

Following creation and output of a Data Member Directory, the mdl field will be equal to the DMD length and the nwa field will equal mda+mdl.

3.4.1.3 Data Member Directory (DMD)

System Table Type: 1

Residence: Data Units

Primary Users: DBM CREATE control statement, Data Member Manager open read (MMØPRD), and close write (MMCLØS) requests.

Description: The DMD identifies all data members which are written to its data unit and contains the word address and length of each Data Member Header (DMH). When a data member is opened for reading, MMØPRD searches the DMD for the name of the data member. If the data member is found, its DMH address and length are used to read the DMH into the Member Control Block. When a data member that was opened to write is closed an entry is made (or updated) for it in the DMD.

Format:

		Data Member Directory	Position Parameter	Common Block
Preface		id	(See system table type 1 Preface)	
		nae		
		nce		
		le		
Body	$entry_1$	entry data		
		. . .		
	$entry_i$	dmn	MDEDMN	/XDBMC/
		mha	MDEMHA	
		mhl	MDEMHL	
		. . .		
	$entry_{nae}$	entry data		

dmn - the data member name
mha - the Data Member Header (DMH) address
mhl - the length of the DMH

Initialization: When initially created the DMD is zeroed and its preface is initialized as follows:

id = IDMD from /XDBMC/ common block
nae = NAEMD from /XDBMC/ common block
nce = zero
le = LENMDE from /XDBMC/ common block

The length in words of the DMD is computed as follows:

LEN = NTSTRT + NAEMD * LENMDE

where NTSTRT is from /XCVT/ common block.

3.4.2 Data Member

A data member is an ordered set of information which resides, in a logically contiguous fashion, on a data unit. The information can be viewed as two subsets of data, (1) user data and (2) non-user or Data Member Manager (DMM) data.

User data are those data which are generated by an Executive or Functional Module and passed to DMM for storage and subsequent retrieval. The form and content of these data base structures is discussed elsewhere.

DMM data are structures which provide information about the form, location, and amount of user data stored as a data member on a data unit. These structures are described in the following paragraphs.

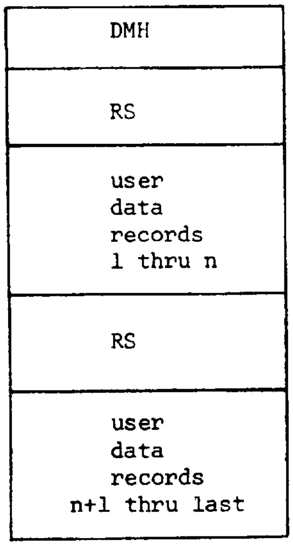
3.4.2.1 Data Member Structure

Residence: Data Units

Primary Users: Data Member Manager (DMM)

Description: A data member is composed of a Data Member Header (DMH), Record Subdirectories (RS), and data records which are addressed using the RS.

Format:



DMH - the data member header is variable length and contains the Master Record Directory (RD) which indexes the RS.

RS - the record subdirectories are variable length with their number and length dependent on the maximum number of records specified by the user when the member was created.

Life Span: The life span of a data member on an external storage device is dependent upon the user's retention of the data unit to which it is assigned.

Initialization: Not applicable

3.4.2.2 Data Member Header (DMH)

Residence: Data Members

Primary Users: Data Member Manager (DMM)

Description: The DMH is the source of quantitative, historic, and reference information for a data member. It consists of a Preface, Format Specification Image, Format Specification Table, and Record Directory. When a data member is opened for writing, the DMH is created as part of the Member Control Block (MCB), Seçtion 3.3.6, and space is reserved for it at the beginning of the data member. As the data member is written, information on number of records written, maximum record length, and number of Record Subdirectories written is stored in the DMH. Closing the data member causes the DMH to be written on the data unit preceeding the other data member data. Subsequent opening of the data member for reading will cause the DMH to be read into the MCB.

EXECUTIVE DATE BASE STRUCTURES

Format:

	Data Member Header	Position Parameter	Common Block
DMH Preface	dmn	MHDMN	/XDBMC/
	len	MHLEN	
	mnr	MHMNR	
	cnr	MHCNR	
	mrl	MHMRL	
	fhl	MHFHL	
	vtl	MHVTL	
	date	MHDATE	
	time	MHTIME	
	rdl	MHRDL	
	nrs	MHNRS	
	fsil	MHFSIL	
	fstl	MHFSTL	
Body	FSI	MHFSI	
	FST		
	RD		

dmn - data member name
len - member header length
mnr - maximum number of data records specified in the open member request
cnr - current number of user records
mrl - maximum record length
fhl - fixed header length
vtl - length of repeated variable trailer
date - date member created in form YY/MM/DD
time - time member created HH.MM.SS
rdl - record directory record length
nrs - number of record subdirectories
fsil - format specification image length
fstl - format specification table length
FSI - format specification image
FST - format specification table
RD - record directory

Initialization: When the DMH is first created, the following fields are initialized:

dmn = NAMA(2) from /XDBMC/ common block
len = MHSFSI + FSIL + FSTL + RDL, where MHSFSI is from /XDBMC/ common block and FSIL, FSTL, and RDL are entries in the data member header
mnr = 10000 if mnr specified open member request is zero; otherwise, unchanged
cnr = 0
mrl = 0
fhl = LENFH where LENFH is computed by MMBFST and is summation of data element lengths of the fixed part of format specification if FSI is non-zero; otherwise zero
vtl = LENRG where LENRG is the summation of the lengths of the elements in the variable part of the record if the FST specifies a variable length formatted record; otherwise zero

date = by IDATE
time = by ITIME
rdl = LENRDB = $(\text{MNR})^{\frac{1}{2}} + 2.99999$
nrs = 0
fsil = LENFSI length of format specification image as determined by MMBFSI
fstl = LENFST length of format specification image as determined by MMBFST

The initialization of FSI, FST, and RD are discussed in their subsections.

3.4.2.3 Format Specification Image (FSI)

Residence: Data Member Headers

Primary Users: Data Member Manager (DMM) open write routines, LØAD, UNLØAD, and UPDATE control statements.

Description: The FSI cannot be described with the tabular presentation used for other data base structures. It is a Hollerith string of ANOPP data type descriptors which are separated by commas. The data types may be grouped using parentheses. Single data types and groups may be prefixed by an integer character string or an asterisk to indicate repetition. The FSI is always terminated with a dollar sign. If the data member which the FSI describes is unformatted then the FSI will be one word of binary zeroes. The FSI is initially created by subprogram MMBFSI from the format specification provided on DMM open write requests (MMØPWD, MMØPWS) and is stored in the Data Member Header. The LØAD, UNLØAD, and UPDATE control statements retrieve it from there for their own use.

Format: Not applicable

Life Span: The FSI is core resident in a MCB when a data member is open. The in-core life span of a particular FSI is, therefore, dependent upon how long the data member remains open to a module.

The life span of the FSI on secondary storage devices is dependent upon the retention period of the file on which its data member and unit reside.

Initialization: Not applicable

3.4.2.4 Format Specification Table (FST)

Residence: Data Member Headers

Primary Users: Data Member Manager (DMM) get and put element routines (MMGETE, MMPUTE)

Description: The FST is an array of element descriptors which specify the format of records contained on, or to be written to, a particular data member. The element descriptors have the following formats:

Single Element Descriptor

A single element descriptor is one word in length and its value is less than seven and not equal to zero. The length of the element is determined as follows:

1. If the element type (value) is greater than zero then the value is used as an index to the NDTCL table in /XCVT/ common (ELEN = NDTCL(VALUE,3)).

2. If the element descriptor value is negative then the value is the absolute value of the element descriptor value and the length is based on the NCPW variable in /XCVT/ (ELEN = (-VALUE+NCPW-1)/NCPW).

Repeated Group Descriptor

A repeated group descriptor is built when the users format requires one or more elements or element groups to be repeated in the record. Nesting of repeated groups is permitted. The repeated group descriptor consists of a three word header, a group of element descriptors of any type, and a two word trailer.

Several subprograms were written to manage the FST for DMM. They are:

1. MMGED - get next element description

2. MMGNEW - determine the number of elements that fit an array of NWDS words

3. MMGNWE - determine the number of words required to write the next NEL elements

4. MMSFEI - reset the FST element index based on the number of words read or written in the current record

Format: The following format represents a user format of the type I, RS, 3(A3,RS),*RD $.

		Table Index	Format Specification Table			Position Parameter	Common Block
Format of Fixed Length Header		1	e_1	1			/XDBMC/
		2	e_2	2			
	Fixed length repeat group	3	rgh_1	22	repeat		
		4	rpt_1	3	group	IRGRPT	
		5	cnt_1	0	header	IRGCNT	
		6	re_1	-3			
		7	re_2	2			
		8	rgt_1	23	repeat group	IRGRTN	
		9	rtn_1	3	trailer		
Format of Variable Length Trailer	variable length repeat group	10	rgh_2	22	repeat		
		11	rpt_2	0	group		
		12	cnt_2	0	header		
		13	re_3	3			
		14	rgt_2	23	repeat group		
		15	rtn_2	10	trailer		

e_1,e_2,re_1,re_2, and re_3 - all Single Element Descriptors having a value greater than -133, less than 7, and not zero

rgh_1 and rgh_2 - repeat group element types with a value equal to IRGME in /XDBMC/ common block. They indicate the beginning of a repeat group header

$rpt_1$and rpt_2 - contain the integer number of times the elements bracketed by the repeat group headers and trailer are to be repeated. rpt_1, is greater than zero indicating the repeat group has a fixed number of repetitions, while rpt_2 is zero indicating an indefinite number

cnt_1 and cnt_2 - are zero and are used in element level processing to control the number of times a repeat group is repeated

rgt_1 and rgt_2 - repeat group trailer element types with a value equal to IRGTE in /XDBMC/ common block. They indicate the beginning of repeat group trailer

rtn_1 and rtn_2 - indices, relative to the beginning of the FST, to their related repeat group header

Initialization: Not applicable

3.4.2.5 Record Directories (RD)

Residence: Data Members

Primary Users: Data Member Manager (DMM) get and put subprograms

Description: The Record Directory is the first level of a two level data record index which provide DMM with a unified approach to random and sequential accessing of fixed format, variable format, and unformatted records. The RD is the index to the second level, the Record Subdirectory (RS), which contains the secondary storage addresses (relative to beginning of data member) (word addresses on CDC CYBER computer system) of the actual data records.

The RD record has a fixed format and, within a data member, a fixed length. However, from data member to data member the length may differ depending on the maximum number of records (MNR) the user has allocated to a member at open time (MMØPWD, MMØPWS). This length is calculated using the following algorithm:

$$LEN = (MNR)^{\frac{1}{2}} + 2.9999$$

where LEN is integer and the result is truncated.

Format:

Record Directory

id
wa_1
⋮
wa_i
⋮
wa_{last}
0

id - RD identifier
wa_i - the addresses of the secondary storage addresses of RS.

Initialization: The R record is zeroed prior to use.

3.4.2.6 Record Subdirectory (RS)

Residence: Data Member

Primary Users: Data Member Manager (DMM) get and put subprograms

Description: The RS is the second level of a two level data record index which contains the secondary storage addresses relative to the beginning of the data member (word addresses on CDC CYBER computer system) of the actual data record.

The RS record has a fixed format and, within a data member, a fixed length.

Format:

Record Subdirectory

id
wa_1
. . .
wa_i
. . .
wa_{last}
nxtwa

id - table identifier
wa_i - addresses of the secondary storage addresses of data records
nxtwa - chain word from current to the next RS record

Initialization: The RS record is zeroed prior to use.

id = IDRS from /XDBMC/ common block.

3.4.3 Data Table Types

Residence: Data Tables reside on the unit specified by the user when the table is built. While the table is open for processing, it resides in Global Dynamic Storage.

Primary Users: EM modules XTB and DBM module TMBLD1 which build Data Tables and DBM module TMTERP which retrieves an interpolated dependent variable from a currently open table.

Description: A data table is a user created table of data available to the functional modules for processing. It is a one-record member having an internal format corresponding to one of the Data Table Structures defined. Currently, only Type 1 data tables have been implemented.

The general table structure consists of a fixed length preface of identical format for all tables and a variable length body of specific format for the individual table type. A Data Table is built through the control statement stream using the TABLE control statement or in a functional module using the DBM table build routines.

When first opened for processing (TMØPN), a table is read into core and its name is entered into the table directory. When closed (TMCLØS) the table remains in core and is logically closed in the directory. Subsequent opens will take place in core via the directory. If a table is altered while it is open in core, it should be closed with a close alter (TMCLSA) so that a copy will be placed on the original member it came from. This is necessary to preserve the integrity of the table under the following conditions: (a) while a table is logically closed in the table directory it can be removed from the directory for one of two reasons: (1) to make room for other tables or (2) because member manager is processing the member for writing; and (b) when the table is removed from the directory, a subsequent open will read a new copy of the member into core and place its name into the table direectory.

Format:

	Data Table	Position Parameter	Common Block
Preface	idtabl	NDTID	/XDTMC/
	type	NDTTP	
	length	NDTLN	
	rsvd		
	rsvd		
	rsvd		
	rsvd		
Body	depends on type	NDTST	

idtabl - data table identifier (8HDATATABL)
type - table type (integer number)
length - length of this table (includes Preface and Body)
rsvd - reserved four words

Initialization: There is no initialization.

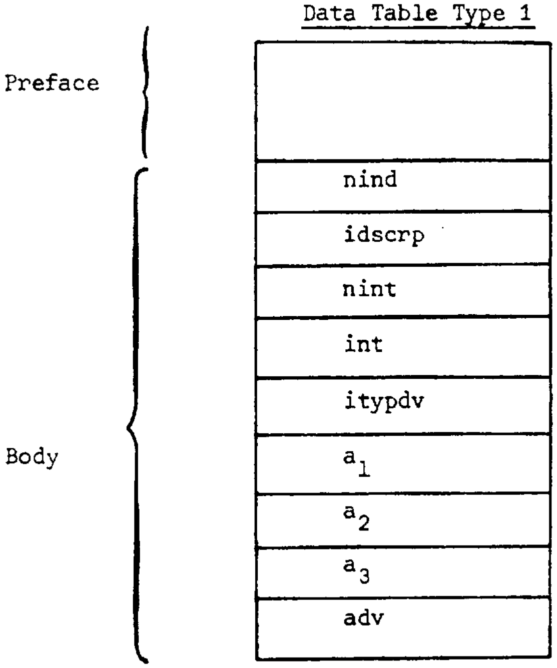
Life Span: A data table remains in core until: (1) it is closed and its space in the table directory is needed to open another table, (2) it is being processed by member manager for writing, or (3) the termination of the current ANOPP run, whichever comes first.

3.4.3.1 Data Table Type 1

Description: For a full description of Data Tables Residence, Primary Users, Life Span, Initialization, see Section 3.4.3. A Type 1 table defines a Data Table body format which specifies interpolation procedures acceptable on this table and the interpolation procedures to be used when an interpolation request is outside the table range of the independent variable.

Format:

Data Table Type 1



(See data table type preface)

nind - number of independent variables in this table .LE.3

idscrp - array of dimension nind*4

idscrp(4*i-3) - format code of ith independent variable
0 - ordered position from 1 to number of elements in independent variable array. In this case a_i does not exist.
1 - integer (I)
2 - real single (RS)
3 - real double (RD)

idscrp(4*i-2) - integer number of ith independent variables

idscrp(4*i-1) - exrapolation procedure to be used if the ith independent variable value supplied is greater than largest value of the ith independent variable in the table
0 - extrapolation not allowed
1 - use closest independent variable value
2 - extrapolate

idscrp(4*i) - exrapolation procedures to be used if the value supplied is less than the smallest value of the ith independent variable
0 - extrapolation not allowed
1 - use closest independent variable value
2 - extrapolate

nint - number of elements in array containing interpolation procedures acceptable on this table

int - array containing one or more integer codes of interpolation procedures acceptable on this table. Acceptable codes are:
0 - no interpolation
1 - linear interpolation

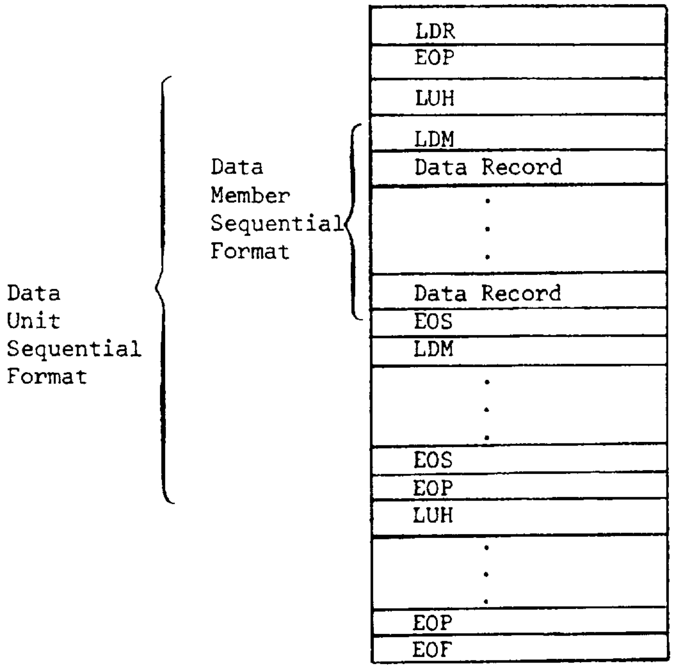
itypdv - format code of dependent variable
1 - integer (I)
2 - real single (RS)
3 - real double (RD)

a_1, a_2, a_3 - one-dimensional arrays of values for the first, second, and third independent variables, if they exist. Values must be arranged in monotonically increasing or decreasing order

adv - nind-dimensional array of dependent variable values such that the first independent variable varies first, the second variable second, and the third, third.

Format:

Sequential Library File Structure



LDR	-	see Subsection 3.4.4.2
LUH	-	see Subsection 3.4.4.3
LDM	-	see Subsection 3.4.4.4
EOS	-	CYBER Record Manager End-of-section
EOP	-	CYBER Record Manager End-of-partition
EOF	-	CYBER Record Manager End-of-file

Life Span: During an ANOPP run an SLF is available for use following its creation by UNLØAD or, if it was existent before the run, at any time via LØAD. A SLF may be made unavailable within ANOPP through use of the DRØP control statement. This, however, will also remove the file from the ANOPP run's operating environment and it cannot be reestablished within ANOPP.

Initialization: See Subsections 3.4.4.2 through 3.4.4.4.

3.4.4 Sequential Library

3.4.4.1 Sequential Library File Structure (SLF)

Residence: ANOPP library files reside on secondary storage devices (rotating mass storage or magnetic tape) and are identified by a file name known to both ANOPP DBM and the host computer operating system.

Primary Users: LØAD (XLD) and UNLØAD (XUN) control statements.

Description: A Sequential Library File is a set of data units, and a complete or partial subset of their data members, which has been converted from random to sequential access file structure. Data units and members within units are unloaded to the SLF, by name, in ascending binary sequence to permit optimization of data unit and member loading. Once created, individual data units or members on a SLF cannot be modified in place without destroying the entire set of units.

On CDC CYBER computer systems the SLF will be recorded using Cyber Record Manager internal blocking (BT=I) and data records will be preceded by a control word (RT=W). This will provide: (1) efficient data transfer to all types of secondary storage devices, (2) maximum data recoverability if errors are encountered when loading from the SLF, and (3) compatibility between CYBER 76 and other CYBER Series computers.

3.4.4.2 Library Directory Record (LDR)

System Table Type: 1

Residence: Sequential Library Files (SLF) and Local Dynamic core storage; the IDX is IDXLDR in /XSLF/ common block.

Primary Users: UNLØAD (XUN) and LØAD (XLD) control statements

Description: The LDR is a type 1 system table created by XUN which contains a sorted list of the names of all data members (with their respective data unit names) that were unloaded to the SLF. The LDR is written as the first record on a SLF by XUN and is subsequently used to control the sequence in which data units and members are unloaded. XLD uses the LDR to insure the existence of data members named on a LØAD control statement prior to processing.

Format:

		Library Directory Record	Position Parameter	Common Block
Preface		id	(See system table type 1 Preface)	
		nae		
		nce		
		le		
Body	Entry_1	entry data		
		. . .		
	Entry_i	dun	LDRDUN	/XSLF/
		dmn		
		. . .		
	$\text{Entry}_{\text{nae}}$	entry data		

dun - data unit name
dmn - data member name

Life Span: Since the LDR is recorded on a sequential library file, its life span outside ANOPP is dependent upon retention of the SLF by the user. Within ANOPP a particular LDR is resident during XUN and XLD processing and is available from sequential libraries.

Initialization: XUNBGN allocates the LDR at the beginning of UNLØAD control statement processing as follows:

1. The number of words initially allocated to the LDR is determined by:

   $$\mathrm{LEN} = \mathrm{NTSTRT} + \mathrm{NAELDR} * \mathrm{LENDRE}, \quad \text{where}$$

   NTSTRT is a variable from /XCVT/ common block, and NAELDR and LENDRE are from /XSLF/ common block.

2. The LDR preface is initialized with the following:

id	=	IDLDR from /XSLF/ common block
nae	=	NAELDR from /XSLF/ common block
nce	=	set to zero
le	=	LENDRE from /XSLF/ common block

3. The body of the LDR is initially set to zero.

3.4.4.3 Library Data Unit Header (LUH)

Residence: Sequential Library Files (SLF)

Primary Users: LØAD (XLD) and UNLØAD (XUN) control statements

Description: The LUH is a subset of information from the Data Unit Header that is necessary to identify and restore the original data unit when it is loaded. It is generated by UNLØAD and indicates the start of a new data unit on sequential library files. The LUH has a fixed record length specified by LENLUH in the /XSLF/ common block.

Format:

Library Data Unit Header	Position Parameter	Common Block
id	LUHID	/XSLF/
dun	LUHDUN	
arflg	LUHAF	
ndm	LUHNDM	

id - LUH Record identifier (Hollerith)
dun - Data Unit Name
arflg - Archived Data Unit Flag
ndm - Number of data members unloaded from the named data unit

Life Span: The life span of the LUH depends on the retention period for the sequential library file on which it resides.

Initialization:

id = IDLUH from the /XSLF/ common block
dun = initialized with the Data Unit Name variable from the related DUD entry
arflg = initialized with the Archive Flag variable from the related DUD entry
ndm = initialized with the number of members to be unloaded from the specified data unit

3.4.4.4 Library Data Member Structure (LDM)

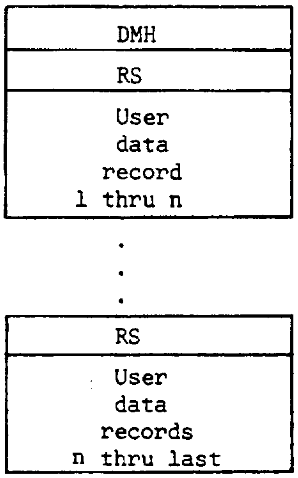
Residence: ANOPP Library Files

Primary Users: LØAD and UNLØAD control statements

Description: The LDM is a copy of a data member on an ANOPP data unit. It contains all of the information contained on the data member including the Data Member Header, Record Directory and Subdirectories.

Format:

Library Data Member Structure



DMH - the Data Member Header is variable length and contains the Record Directory (RD) which indexes the RS.

RS - Record Subdirectories are variable length with their number and length dependent on the maximum number of records specified by the user when the member was created.

Life Span: The life span of a particular LDM is dependent upon the retention period for the file on which it resides.

Initialization: Not applicable

3.4.5 Executive Management System Reserved Units

The Executive Scratch unit and Data unit are created by XBSDBM which initializes the Data Base Management System. They exist throughout ANOPP execution.

3.4.5.1 Executive Management System Scratch Unit

Residence: The Executive scratch unit resides on a random access secondary storage device and is identified by the name NXUNIT from /XCS/ common block.

Primary Users: The Executive Scratch Unit is used by XRT (The Primary Edit Phase), XCA (The Secondary Edit Phase), XCSP (The Control Statement Processing Phase), and UPDATE.

Description: The Executive Scratch Unit is a set of Mxxx and Uxxx data members (described in the following sections). It contains a Data Unit Header (DUH) and a Data Member Directory (DMD) which contain the information necessary to access and add members. The format of a data unit in general are described in Section 3.4.1.1.

3.4.5.1.1 Mxxx Member

Residence: The Mxxx Member resides on the Executive Management System (EM) scratch unit XSUNIT.

Primary Users: EM modules XRT and XCA which construct Mxxx Members and EM modules XCSP and XMERR which use Mxxx members for processing.

Description: xxx is a display code of integers 001 through 999. M001, the root member, is built from the control statement images in the Primary Input Stream. M001 is created during the Primary Edit Phase by XRT and contains the Primary Control Statement Set in executable form.

Mxxx, xxx.GT.001, is created during the Secondary Edit Phase in XCSP when a CALL CS is encountered and contains a Secondary Control Statement Set in executable form. The number xxx is assigned sequentially from 002 as each CALL CS is encountered for the first time.

An Mxxx Member is made up of N unformatted, variable length records. The first N-1 records are Mxxx Control Statement Records (CS Records), one CS Record for each CS image encountered in the input stream, and the last record is the Mxxx Label Record.

A CS Record is the executable form of one complete CS image supplied by the user in the Primary or Secondary Input Stream. The CS records are sequenced in the order corresponding to the occurrence of the CS in the Primary Input Stream.

The Label Record contains an entry for each labelled control statement in the CS stream and specifies all label names and their corresponding CS record location. The Label Record is variable length depending on the number of label entries. However, the presence of the Preface always insures a Label Record length of .GE.1.

Format:

	Mxxx Member
Record$_1$	first CS Record
	⋮
Record$_{N-1}$	last CS Record
Record$_N$	Label Record

where N is the number of records on Mxxx

Format:

		Control Statement Record	Position Parameter	Common Block
Preface		len	KLCS	/XCS/
		lab	KCSLAB	
		nam	KCSN	
		nimg	KNIMG	
		nodb	KNØDB	
		stodb	KØDB	
		mem	KMEM	
		stimg	KIMG	
		. . .		
		endimg		
Body	$ØDB_1$	entry data		
		. . .		
	$ØDB_i$	type value		
		. . .		
	$ØDB_{nodb}$	entry data		

Preface - includes word zero through the last word of the CS image. The length of the preface varies with the number of words required for the CS image. The maximum preface length is 7+ (maximum cards allowed per CS)* (number of words per card image) which is (7 + 5 * 10) or 57.

len - number of words in this record

lab - name of CS label (A8) if this CS had a label; otherwise blank

nam - CS name (A8)

nimg - number of words in the CS image

nodb - number of Operand Description Blocks (ØDB) in the CS record

stodb - start of the ODB entries relative to the start of the CS Record.

mem - Contains one of the following entries depending on the CS name:

1. if this is a CALL CS - name of the Mxxx Member (A8) which contains the Secondary CS Set ready for the CS Processing Phase.
2. if this is an UPDATE CS or TABLE CS with a source = * - name of the Uxxx Member (A8) which contains the card image input
3. if not 1 or 2, then blanks

stimg - CS image in A8 format. Length of the CS IMAGE is (number of cards in this CS) * (number of words per card image)

ending - end of CS image

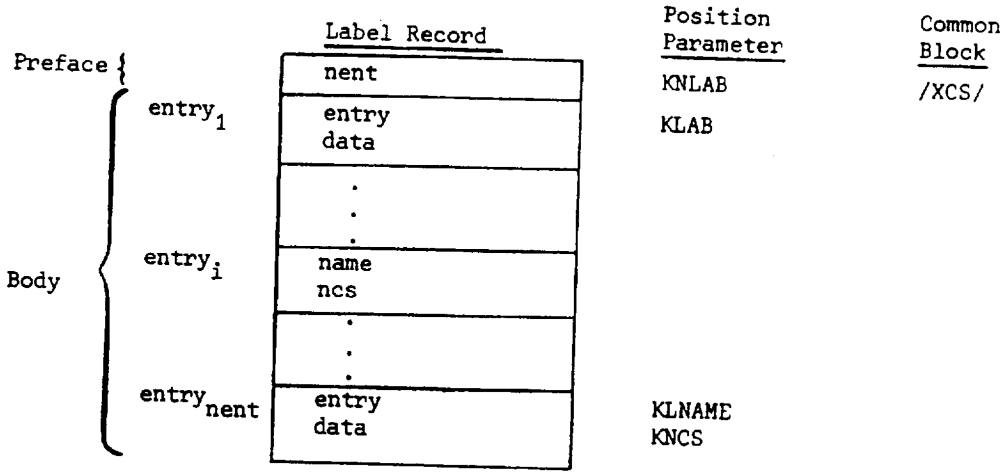
Body - the body is made up of Operand Description Block (ODB) entries. Each field or delimiter (other than comma or blank) following the CS name on the CS card image(s) has an ODB entry in the order encountered on the CS card image

ODB_i - Operand Description Block entry - variable length entry for each field encountered in the CS. The first word contains the type of the field and the following word(s) contain(s) the value of the field

Type - ANOPP integer type code of field

Value - field value - length implied by type code

Format:



nent - word preface contains the number of entries in this record

name - label name (A8)

ncs - number record within the Mxxx member of the CS with this label within the Mxxx member.

Life Span: The Mxxx Members, built during the Primary or Secondary Edit Phase of ANOPP, exist during ANOPP execution.

Initialization: There is no initialization. Once the Mxxx Member is created, it is not altered.

3.4.5.1.2 Uxxx Member

Residence: The Uxxx Member resides on the Executive Management Scratch Unit XSUNIT.

Primary Users: EM module XRT which constructs the Uxxx Member and EM module XCSP which uses the Uxxx member for processing. The TABLE and UPDATE control statements.

Description: The Uxxx Member is created during the Primary Edit Phase by XRT whenever an UPDATE or TABLE control statement with a source = * field is encountered in the Primary Input Stream and contains the UPDATE or TABLE input which immediately follows the corresponding control statement in the Primary Input Stream. This input, in card image format (10A8), is to be used as source input during execution of the corresponding UPDATE or TABLE control statements.

xxx is a display code of integers 001 through 999. The number xxx is assigned sequentially from 001 as each TABLE or UPDATE control statement is encountered during the Primary Edit Phase.

Format:

Uxxx Member

Record$_1$	card image
	. . . . .
Record$_N$	card image

where N is the number of input card images on the Uxxx Member.

Initialization: There is no initialization. Once the Uxxx Member is created, it is not altered.

3.4.5.2 Data Unit

Residence: The DATA unit resides on a random access secondary storage device and is identified by the name NDATA from /XCS/ common block.

Primary Users: XRT (the Primary Edit Phase) builds the members on DATA and DBM modules access members built on DATA.

Description: DATA contains Data Unit Header (DUH) and a Data Member Directory (DMD) which contain information necessary to access and add data members. The DUH and DMD are followed by the data members which are built during the Primary Edit Phase (XRT) whenever a DATA control statement is encountered. The members are built in card image format from the card images which follow the DATA control statement in the input stream until an END* control statement is encountered. The format of data unit in general is described in Section 3.4.1.1.

Format:

DATA Unit

DUH
DMD
members

DUH	-	data unit header
DMD	-	data member directory
members	-	card image format

Life Span: DATA unit exists throughout ANOPP execution.

Initialization: When initially created by XBSDBM, DATA contains only the DUH and DMD. Their initial settings established by XCTDU are described in Sections 3.4.1.2 and 3.4.1.3, respectively.

3.5 EXECUTIVE MANAGEMENT SYSTEM

3.5.1 Overview

The Executive Management System (EM) controls the execution of ANOPP from beginning to end. The following tasks are required to accomplish this control:

1. Perform initialization requirements for the Executive Management System (EM), the Data Base Management System (DBM), the Dynamic Storage Management System (DSM), and the General Utilities.
2. Validate the required user supplied set of control statements which defines the execution sequence.
3. Direct the order of execution dynamically via the required set and the optional set(s) of control statements supplied by the user.
4. Control the execution of Functional Modules (F.M.) by transferring execution control to the specified F.M. and by insuring the integrity of the ANOPP system environment upon completion of the F.M.
5. Direct the action to be taken upon encountering a non-fatal error during execution of ANOPP.
6. Validate the optional set(s) of control statements which define secondary execution sequence(s) to be performed.
7. Terminate ANOPP when all required tasks have been accomplished.
8. Abort ANOPP if a fatal error occurs during the performance of the above tasks.

ANOPP is controlled by a set of control statements called the Primary Input Stream. The Primary Input Stream consists of an optional initialization control statement and a required execution section which defines the execution sequence. The ANOPP control statement allows the user to define selected ANOPP initialization parameters. The execution section begins with a STARTCS control statement followed by the control statements defining the execution sequence and ends with an ENDCS control statement.

Additional control statement sets may define secondary execution sequences to be dynamically executed via a CALL control statement. The additional control statement set is called a Secondary Input Stream and resides as a data member in card image format. The Secondary Input Stream may be prepared prior to the current ANOPP execution or it may be created within the execution sequence prior to the corresponding CALL control statement.

Executive Monitor (XM) is the driver module for the Executive Management System. XM is also the main FORTRAN program for ANOPP and remains in core at all times.

The Executive Management System is composed of eight execution phases which are controlled directly or indirectly by the XM driver module. The following execution phases correspond to the eight Executive Management System tasks previously stated:

1. Initialization Phase
2. Primary Edit Phase
3. Control Statement Processing Phase
4. Functional Module Processing Phase
5. Error Processing Phase
6. Secondary Edit Phase
7. Normal Termination Phase
8. Error Termination Phase

3.5.2 Control Statements

A control statement is one or more cards or card images which defines a particular action to be performed by the ANOPP Executive Management System. A set of control statements is one or more statements which are ordered sequentially to define an execution sequence. et is executed sequentially from the first control statement through the last control statement in the set. The sequential flow may be altered, however, by special control statements which transfer execution control to a specified control statement in the same set or the beginning of another set. There are two types of control statement sets, the Primary Input Stream and the Secondary Input Stream.

3.5.2.1 Primary Input Stream

The Primary Input Stream is a required set of input cards. It consists of an optional initialization control statement (CS), the ANØPP CS, and a required execution section. The ANØPP CS allows the user to define initialization values for selected executive module parameters. If present, it must be the first CS in the Primary Input Stream. If omitted, all parameters are initialized according to predefined installation values. The execution section begins with a STARTCS control statement followed by control statements defining the execution sequence and ends with an ENDCS control statement. If the ANOPP CS is not present, STARTCS is the first card in the Primary Input Stream.

3.5.2.2 Secondary Input Stream

A Secondary Input Stream is an optional set of card images which resides as a data member in card image (CI) format. A CALL in the Primary Input Stream brings into execution a Secondary Input Stream which may also contain a CALL. There is no limit to the number of nested CALL control statements. There is no required first or last control statement (CS) in a Secondary Input Stream. The execution sequence begins with the first CS and ends with the last CS. However all control statements or control statement forms defined for ANOPP are not available for usage in a Secondary Input Stream. The invalid control statements are generally those which require card input to immediately follow the CS. An example is the DATA control statement (see Section 3.5.2.4.7). All control statements of this type require the END* card (see Section 3.5.2.4.11) to terminate the input and are valid in the Primary Input Stream only. The ANOPP, STARTCS and ENDCS control statements are also invalid in the Secondary Input Stream. The validity of each CS with respect to usage is given in the specific CS description section (see Section 3.5.2.4).

3.5.2.3 General Description

3.5.2.3.1 Format

The general format of a control statement (CS) is shown below:

$$\text{label}_{\not b}\text{csname}_{\not b}\text{op \$ comments}$$

where:

label - an optional 1-8 character alphanumberic name tag which will be associated with this directive. A label is allowed on any control statement except ANØPP, STARTCS, END*, and RETURN.

$\not b$ - a comma or a blank

csname - a valid 1-8 character alphanumeric control statement name

op - operand field(s) as appropriate to particular control statement

\$ - the end-of-data character which indicates completion of a CS. The remainder of the card may be comment.

3.5.2.3.2 Valid Control Statement Names

Valid control statement names recognized by the ANOPP system are shown below:

ANØPP
ARCHIVE
ATTACH
CALL
CØNTINUE
CREATE
DATA
DETACH
DRØP
ENDCS
END*
EXECUTE
GØTØ
IF
LØAD
PARAM
PRØCEED
PURGE
RETURN
SETSYS
STARTCS
TABLE
UNLØAD
UPDATE

3.5.2.3.3 Free-Field Form

A CS directive is free-field form on a card image in columns 1-80. The fields (including label and CS name) may begin in any column. Blanks or commas may be used freely between fields for spacing and are ignored.

3.5.2.3.4 Comments

A comment may follow the end-of-data character ($) on a card image and extend through column 80. A comment card is a card image (other than a CS continuation card) with the first non-blank character, other than a comma, being a $. The remainder of the card is processed as a comment.

3.5.2.3.5 Continuation

A CS may require more than one card image for completion. A maximum of 5 continuation cards are allowed with the end-of-data ($) character appearing on the last card image required. A Hollerith field type may be continued to column 1 of the continuation card image; however, the nH portion which identifies it as Hollerith must not be split between two card images. No other field types may be split between two card images. Comments may not be continued (except as multiple comment cards).

3.5.2.4 Specific Descriptions

On the following control statements (CS) an optional field is enclosed in [ ]. The symbol b̸ represents a blank. The alphabetic letter is slashed when appearing on a card image (e.g., alpha Ø, numeric 0, alpha Ƶ, numeric 2).

3.5.2.4.1 ANØPP

Purpose: The ANØPP control statement is optional and provides for user-specified values to be used for executive system parameters during the ANOPP run instead of system defined default values. If used, it must be the first control statement in the Primary Input Stream immediately preceding the STARTCS control statement.

Format:

$\text{ANØPP}_{\not b}\text{keyword}_1\text{=value}_1[_{\not b}\ldots_{\not b}\text{keyword}_n\text{=value}_n]\ \$$

keyword - the name of the parameter whose default value the user wishes to override or replace. A list of valid keywords with their corresponding acceptable replacement values and system default values is given in the ANOPP Keyword Table (see Table 1).

value - the value which is to replace the system default value for the corresponding keyword. A label field is not allowed on the ANØPP statement.

Examples:

```
ANØPP JECHØ=.TRUE. $
ANØPP,JLØG=.TRUE. JECHØ =.TRUE. $
ANØPP NLPPM = 40,LENGL=30000,JLØG = .FALSE.$
```

Restriction: ANOPP is valid only as the first CS in the Primary Input Stream.

KEYWORD	DESCRIPTION	RANGE OF ACCEPTABLE VALUES	DEFAULT PARAMETERS		
			Residence	Name	Value
JECHØ	Print card images from the primary input stream or from a member (via a CA11 CS) as they are validated in Primary Edit Phase and Secondardy Edit Phase JECHØ = .TRUE. then print card images JECHØ = .FALSE. then do not print card images	.TRUE. .FALSE.	/XSPT/	JECHØ	.FALSE.
JLØG	Print card images of control statements as executed in the Control Statement Processing Phase JLØG = .TRUE. then print card images JLØG = .FALSE. then do not print card images	.TRUE. .FALSE.	/XSPT/	JLØG	.TRUE.
LENGL	Length of global dynamic storage initialized for this ANOPP run	Integer $_{10} \geq 3000_{10}$	/XCVT/	LENGL	$20{,}000_8$
NAETD	Number of initially allocated entries in the Table Directory	Integer $\geq$ 1	/XDTMC/	NAETD	10_{10}
NAEUD	Number of initially allocated entries in the United Directory	Integer $>$ 4	/XDBMC/	NAEUD	25_{10}
NLPPM	Number of lines per printed pages to be used for all ANOPP printed output	Integer $15_{10} <$ NLPPM	/XCSFM/	NLPPM	48_{10}
NØGØ	PRIMARY EDIT ONLY NØGØ = .TRUE. then terminate ANOPP after Primary Edit Phase NØGØ= .FALSE. then do not terminate ANOPP after Primary Edit Phase	.TRUE. .FALSE.	/XCS/	NØGØ	.FALSE.

Table 1. ANOPP Control Statement Keyword Table.

3.5.2.4.2 ARCHIVE

Purpose: The ARCHIVE control statement permanently prohibits any request to write on the named unit or units.

Format:

$$[\text{label}_{\cancel{b}}]\text{ARCHIVE}_{\cancel{b}}\text{du}_1[\ldots{}_{\cancel{b}}\text{du}_n]\ \$$$

label - label name

du - name of the data unit to be archived

The specified unit(s) are permanently archived and no future writes to the unit(s) are permitted. The archive indicator is written to the unit thus prohibiting output to the unit(s) even in subsequent ANOPP runs.

Examples:

```
LAB,ARCHIVE U1, U2 U10, U50 $
ARCHIVE,U50 $
```

3.5.2.4.3 ATTACH

Purpose: The ATTACH control statement attaches data units to the internal system which were previously created on direct access files and are presently assigned in the external system.

Format:

$$[\text{label}_{\cancel{b}}]\text{ATTACH}_{\cancel{b}}\text{du}_1[/\text{efn}_1/][..,\text{du}_n[/\text{efn}_n/]]\ \$$$

label - label name

du - name of data unit to be attached

efn - name of the external file associated with du

An entry is made in the Unit Directory for each data unit named (du) and the external file name specified (efn) is associated with the unit.

Examples:

```
ATTACH  UNIT1/M0150/, UNIT2/U001/ $
ATTACH  UNIT2/EFN/ $
```

Restriction: Data units appearing on the ATTACH CS must have appeared on a DETACH CS in this or a previous ANOPP execution.

3.5.2.4.4 CALL

Purpose: The CALL control statement executes a set of control statements which have been previously created in card image format (CI) on a data member of a data unit. Prior to execution the CS images will be altered according to specified value replacements.

Format:

$$[\text{label}_{\not{b}}]\text{CALL}_{\not{b}}\text{du(dm)}\ [,\text{oldvalue}_1\text{=newvalue}_1,\ldots,\text{oldvalue}_n\text{=newvalue}_n]\$$$

label - label name

du - the name of the data unit containing dm

dm - the name of the data member which is composed of the set of control statements to be executed

oldvalue - the field value which when encountered on a CS card image is to be replaced with the corresponding "new value". The old value may be any valid field type (I, RS, RD, L, AO, LO, N, A) recognized by the XCR module when cracking a card image.

newvalue - the field value which will replace the "old value". The new value may be any valid field type (I, RS, RD, L, AO, LO, N, A) described in Section 3.9, Table 1. The type of this old value does not have to be the same as the type of the new value; any field value may be replaced by any other field value.

Each control statement (one or more card image records) on the member dm will be searched for fields which match the "old value" parameters specified on the CALL CS. If a field is found to match an "old value" then it is replaced with the corresponding "new value". The types of "old value" and "new value" may or may not agree.

After all replacements have been successfully accomplished, the set of control statements are executed sequentially beginning with the first CS in the dm member. Of course, execution flow sequence within the secondary control stream may be altered by a CS such as GØTØ, IF, or another CALL.

Upon completion of the set of CALLed control statements, execution will continue with the CS immediately following the CALL CS.

Examples:

```
LABEL CALL UN1(DM2),ANAME=BNAME,10=15 $
CALL,UN2(DM2),3HABC=6HABCDEF,10=15.5 $
CALL UN3(DM3),.TRUE.=.FALSE.,+=*,*=NAME $
```

Restrictions: du (dm) was previously created during this ANOPP run or on a previous ANOPP run with a fixed format of CI (the normal procedure for creating du (dm) is via the DATA or UPDATE CS).

3.5.2.4.5 CØNTINUE

Purpose: The CØNTINUE control statement allows for a no action step in execution. It is used mainly with a GØTØ to allow processing flow alteration.

Format:

[label$_{\not b}$]CØNTINUE $

label - label name

Examples:

```
LABEL1 CØNTINUE $
```

3.5.2.4.6 CREATE

Purpose: The CREATE control statement defines an empty data unit on a direct access storage device for subsequent output of members.

Format:

[label$_{\not b}$]CREATE$_{\not b}$du$_1$[/efn$_1$/][...,du$_n$[/efn$_n$/]] $

label - label name

du - the name of the data unit to be associated with the data to be output on efn for this ANOPP run

efn - the name of a file, known to the user, which is defined in the external system. If omitted, a scratch file name is assigned.

Examples:

```
[LABEL]CREATE UNIT1,UNIT2/EFN2/,UN3/E3/ $
 CREATE UNIT2 $
```

Restrictions: The du and efn cannot respectively be the same as another data unit name or external file name currently entered in the data unit directory.

3.5.2.4.7 DATA

Purpose: The DATA control statement creates a data member on data unit DATA.

Format:

label$_{\not b}$ DATA$_{\not b}$DM=dmname $

label - label name

dmname - the name of the data member to be created on data unit DATA.

Data which is to form the data member, dmname, must immediately follow the DATA CS directive in the Primary Input Stream. The data card images must be terminated by an END* CS following the last card image to be used as data.

Dmname will be written on data unit DATA. Dmname will have a fixed format of 10A8 which corresponds to a card image. Each record on dmname will correspond to a single card image read from the Primary Input Stream. Any valid FORTRAN character is acceptable on the card image in any sequence except as follows: beginning with the first non-blank character, the first four (4) characters on a card image must not be END* as this is recognized as the DATA CS input terminator (i.e., a card image beginning with END*AB is not an acceptable data card image).

Examples:

```
LABEL DATA DM = DMEM $
1     2    3       4 ... 50
1.5   2.1  6.0    .TRUE. ... 40
END* $
```

The data member DMEM will consist of two records of fixed format 10A8. Record 1 will contain the card image:

```
1     2     3     4     ...   50
```

Record 2 will contain the card image:

```
1.5   2.1  6.0  .TRUE. ...  40
```

Restrictions: The same name must not appear on another DATA CS within the same ANOPP run.

3.5.2.4.8 DETACH

Purpose: The DETACH control statement removes a data unit from the set of data units known to the ANOPP run. The status of the external file associated with this data unit is left unchanged.

Format:

$$[\text{label}_{\not{b}}]\text{DETACH}_{\not{b}}\text{du}_1[\ldots,\text{du}_n]\$$$

label - label name
du - name of the data unit known to the ANOPP run

NOTE: Unless the data unit resides on a scratch file the external file is available for subsequent ATTACH CS statements.

Examples:

```
L1  DETACH  AIRCR,AIRPRTS,WCØN $
    DETACH MSC $
```

3.5.2.4.9 DRØP

Purpose: The DRØP control statement drops a sequential library file assigned to this job from the external system. The sequential library file is disassociated with the current execution.

Format:

$$[\text{label}_{\not{b}}]\text{DRØP}_{\not{b}}/\text{sefn}_1/[\ldots,/\text{sefn}_n/]\ \$$$

label - label name
sefn - name of the sequential library files to be dropped.

Examples:

```
END  DRØP  /TAPE1/,/TAPE2/,/FILE1/ $
DRØP  /FLE/  $
```

Restrictions: Sequential library file names (sefn) must be unique.

3.5.2.4.10 ENDCS

Purpose: The ENDCS control statement when processed terminates the ANOPP run. There must be only one ENDCS per run, and it must be the last control statement in the Primary Input Stream. It is the only control statement which will terminate an ANOPP run.

Format:

$[\text{label}_{\not b}]\text{ENDCS }\$$

label - label name

Examples:

```
ENDCS $
STØP ENDCS $
```

Restriction: ENDCS is valid in the Primary Input Stream only.

3.5.2.4.11 END*

Purpose: The END* control statement indicates the end of the card image input stream required by the previous control statement, DATA, UPDATE or TABLE.

Format:

```
END* $
```

Examples: See DATA CS, UPDATE CS, and TABLE CS for specific examples.

Restriction: END* is valid in the Primary Input Stream only. A label field is unacceptable on an END* CS.

3.5.2.4.12 EXECUTE

Purpose: The EXECUTE control statement defines a set of alternate names and executes a specified functional module.

Format:

$[\text{label}_{\not b}]\text{EXECUTE fmname}_{\not b}[\text{refname}_1\text{=altname}_1,\ldots,\text{refname=altname}_n]\ \$$

label - label name
fmname - the name of the functional module which is to be executed.
refname - the reference name which has a corresponding name
altname - the alternate name corresponding to the reference name

A set of reference names (refname) and corresponding alternate names (altname) is established (both refname and altname are valid name fields). The functional module is placed in execution immediately. During the execution, the set of alternate names is available for retrieval by the functional module or by an executive system module (for full explanation, see ANOPP Parameter Maintenance Functions (PMF) utilities, Member Manager Subprogram input arguments, and Table Manager subprogram input arguments).

Examples:

```
L1  EXECUTE JET A1=UAB,B=D $
EXECUTE  JET $
```

Restriction: EXECUTE is valid in the Primary or Secondary Input Stream. fmname must refer to a functional module installed and recognized by the ANOPP executive system.

3.5.2.4.13 GØTØ

Purpose: The GØTØ control statement allows an alteration in the flow of control statement processing. Processing will continue at the control statement containing the label specified.

Format:

[label$_{\not b}$]GØTØ$_{\not b}$labnam $

labnam - the label name of the control statement at which processing should continue

Examples:

```
NAME    GØTØ   NAME2  $
GØTØ    NAME $
```

Restriction: Labname must be in the label field of a control statement which is within the same set of control statements as this GØTØ.

3.5.2.4.14 IF

Purpose: The IF control statement permits an alteration in the flow of processing if a specified condition exists. The value of a user parameter is compared with the value of either another user parameter or a constant. If the comparison results in a true condition then processing continues with the control statement having the specified label; otherwise, processing continues with the next control statement.

Format:

$$[\text{label}_{\not{b}}]\text{IF paramname}_{\not{b}}\begin{array}{c}\text{logical}\\ \text{operator}\end{array}\left\{\begin{array}{l}\text{paramname}_2\\ \text{numerical constant}\\ \text{logical constant}\\ \text{string constant}\end{array}\right\}\ \text{GØTØ}_{\not{b}}\text{labnam \$}$$

label - label name

paramname_1 - name of a user parameter whose value is to be compared with the value following the logical operator

logical operator - a logical operator used in comparison of the two values. Any logical operator is valid for comparing values which are type integer, real single precision, or real double precision. The operators .EQ. and .NE. are valid for values of types logical and character string

$\left.\begin{array}{l}\text{paramname}_2\\ \text{numerical constant}\\ \text{logical constant}\\ \text{string constant}\end{array}\right\}$ - the second value for comparison

labnam - the label of the control statement at which processing should continue if the comparison of the two values results in a true condition.

Examples:

```
LABEL  IF (A .GE. B) GØTØ LABEL1 $
       IF (D .EQ. .TRUE.) GØTØ LABEL1 $
       IF (F .EQ. 6HFVALUE) GØTØ LABEL $
```

Restriction: Labnam must be in the label field of a control statement which is within the same set of control statements as this IF. The type of the second value must agree with the value type of paramname_1. If the type is character string then the number of characters in the two strings must be equal.

3.5.2.4.15 LØAD

Purpose: The LØAD control statement loads data units from a sequential library file which has been assigned to the run through the external operating system's control language. This sequential library file must have been created by an UNLØAD CS by the current or a previous ANOPP run. This CS provides selective loading and/or renaming of data units stored in a sequential library file.

Format:

$[\text{label}_{\not b}]\text{LØAD/sefn/ \$}$

or

$[\text{label}_{\not b}]\text{LØAD/sefn/}[\text{dus}_1,\ldots,\text{dus}_n]\text{\$}$ where

dus has the form:

$\text{du}\big[[\text{/efn/}] = \text{odu}\ [(\text{d}_1,\ldots,\text{d}_n)]\big]$

d_i may have either of the forms:

dm_1

$\text{dm}_{\text{new}} = \text{dm}_{\text{old}}$

label - label name

sefn - the sequential external file name of the library file from which data units are to be read

du - the name of the data unit which is to be loaded

efn - the external file name

odu - the name under which the data unit was unloaded. If not specified, it is assumed to equal du

dm - the name of a data member on the odu which is to be loaded from the library. If a member list is not specified, all data members on odu du are loaded from the library

dm_{new} - new name of data member

dm_{old} - name data member was known by

Examples:

```
LØAD/TAPE1/ $
LØAD/TAPE1/UNIT1/EFN1/ (MEM1,MEM2) $
LØAD/TAPE1/UNIT2,UNIT3/EFN3/=OUNIT3 $
```

Restriction: du and efn must not be in the current data unit directory. The name of the data unit to be loaded may not be XSUNIT or DATA. If duplicate data unit names appear on the left of the = , only one occurrence may specify the optional external file name (/efn/).

If data unit name stands alone, as in the form

```
LØAD/LFN1/DUN1 $  or
LØAD/LFN1/DUN1/EFN1/ $
```

then the data unit name (DUN1) may not appear on the right of the = again.

There may be no duplicate data unit and data member name combinations on the right of the = .

3.5.2.4.16 PARAM

Purpose: The PARAM control statement establishes a user parameter or changes the value of an already existing user parameter.

Format:

$$[\text{label}_{\not{b}}]\text{PARAM}_{\not{b}}\text{paramname}_1=\left\{\begin{array}{l}\text{numerical constant}\\ \text{string constant}\\ \text{logical constant}\\ \text{paramname}_2\end{array}\right\}\ \ldots\ \$$$

or

$$[\text{label}_{\not{b}}]\text{PARAM}_{\not{b}}\text{paramname}_1=\text{paramname}_2\left\{\begin{array}{c}+\\ -\\ {}_{\not{b}}*_{\not{b}}\end{array}\right\}\begin{array}{l}\text{numerical}\\ \text{constant}\end{array}\quad \ldots\ \$$$

label - label name

paramname$_1$ - a valid ANOPP name of the user parameter for which a value is to be established or changed

paramname$_2$ - a valid ANOPP name of a user parameter for which a value has already been set

In the first form, paramname$_1$, will be given the value following the equals sign. If paramname$_2$ is specified, the current value of the user parameter paramname$_2$ will be used.

In the second form an algebraic operation will be performed on two values which must be of the same type and the result will become the value of paramname$_1$. The current value of paramname$_2$ must be of the same type as the numeric constant. If paramname$_1$ is the name of a previously established user parameter (via a PARAM CS or an XPUTP call), its value will be changed. The types of the old and new values may or may not agree.

If paramname$_1$ is not the name of a previously established user parameter, then a new user parameter is established and will remain available throughout the ANOPP run. A user parameter once established is never deleted from the set of known user parameters.

Examples:

```
PARAM  A=.5,B=A+10,C=D*2 $
PARAM  F=C $
```

3.5.2.4.17 PRØCEED

Purpose: The PRØCEED control statement allows for processing to continue at a specified point after an ANOPP error has occurred. It is used mainly in conjunction with the system parameter JCØN which may be set via the SETSYS control statement. When a control statement cannot process to its normal completion due to a non-fatal error condition, processing will continue with the first encountered PRØCEED control statement if system parameter JCØN is set to .FALSE. . The PRØCEED, when encountered for execution by normal processing, is a no action step.

Format:

$$[\text{label}_{\not b}]\text{PRØCEED}\ \$$$

label - label field

Examples:

```
ERR1  PRØCEED $
      PRØCEED $
```

Restriction: Upon encountering a non-fatal error, the control stream is searched for a PRØCEED statement. CALL statements encountered in this search are not expanded.

3.5.2.4.18 PURGE

Purpose: The PURGE control statement removes a data unit (or units) from the set of data units known to the ANOPP Data Base Manager and also from the external operating system.

Format:

$$[\text{label}_{\not b}]\text{PURGE}_{\not b}\text{du}_1[\ldots,\text{du}_n]\ \$$$

label - label name
du - name of the data unit to be purged

Examples:

```
APP  PURGE  UN1,UN2,UN4,UN5 $
PURGE  UN6 $
```

3.5.2.4.19 RETURN

Purpose: The RETURN control statement indicates processing of a Secondary Input Stream is complete and allows return to the calling Primary or Secondary Input Stream.

Format:

```
        RETURN $
```

RETURN is created internally during the Secondary Edit Phase (see Section 3.5.4.6) to indicate completion of the Secondary Input Stream. Upon processing a RETURN, execution will continue with the control statement following the CALL which initiated the execution of the Secondary Input Stream.

Examples: Not applicable

Restriction: RETURN is not allowed in the primary or secondary input stream provided by the user. It is simulated during the secondary edit phase for internal control only.

3.5.2.4.20 SETSYS

Purpose: The SETSYS control statement sets the value of a user system parameter.

Format:

$[\text{label}_{\not b}]\text{SETSYS}_{\not b}\text{sysparam}_1\text{=value}_1[,\ldots,\text{sysparam}_n\text{=value}_n]$ \$

label - label name

sysparam - the name of a user system parameter for which a value is to be set. A list is found in Table 2. SETSYS System Parameters Table.

value - the value to which the corresponding user system parameter is to be set. The type of value must be valid and within range for the corresponding user system parameter

A user system parameter may be set several times during the ANOPP run via a SETSYS CS. All user system parameters have initial value settings determined during the ANOPP Initialization Phase. The initial value is the default value defined by the ANOPP system

SYSTEM PARAMETER NAME	DESCRIPTION	TYPE/RANGE	DEFAULT VALUE
JCØN	Controls Executive Managers action when an error has been detected in processing an ANOPP control statement. JCØN = .TRUE. indicates execution will continue with the next CS. JCØN = .FALSE. indicates execution will continue with the next PRØCEED CS. If a PRØCEED is not encountered then the ENDCS CS will be executed for a normal termination.	Logical .TRUE. .FALSE.	.FALSE.
JECHØ	Controls printing of the CS card images upon validation in the Primary and Secondary Edit Phases. All of the Control Statements in the Primary input stream are edited for errors before execution of the first CS (Primary Edit Phase). All of the control statements in a data member which is called into execution via the CALL CS are edited for errors before execution of the first CS (Secondary Edit Phase). JECHØ = .TRUE. indicates the CS images are to be printed JECHØ = .FALSE. indicates the CS images are not to be printed	Logical .TRUE. .FALSE.	.FALSE.
JLØG	Controls printing of the CS card images upon execution in the ANØPP Control Statement Processing Phase. JLØG = .TRUE. indicates the CS images are to be printed JECHØ = .FALSE. indicates the CS images are not to be printed	Logical .TRUE. .FALSE.	.TRUE.

Table 2. SETSYS System Parameters

independent of user control. However, the default value of several system parameters may be set by the user during the Initialization Phase by the ANOPP CS.

Examples:

```
SETSYS  JECHØ=.TRUE. $
ABC  SETSYS JCØN=.TRUE.,JLØG=.TRUE. $
```

3.5.2.4.21 STARTCS

Purpose: The STARTCS control statement indicates the beginning of control statements in the Primary Input Stream. It is required for each ANOPP run. It immediately follows the ANOPP control statement, if present; otherwise, it must be the first card in the ANOPP Primary Iuput Stream.

Format:

```
        STARTCS $
```

Examples:

```
STARTCS $
```

Restrictions: STARTCS is valid in the Primary Input Stream only. A label field is not allowed on the STARTCS.

3.5.2.4.22 TABLE

Purpose: The TABLE control statement builds a data table on a specified member, or subsequent use by Table Manager, from table description input cards.

Format:

```
        [labelƀ]TABLEƀdu(dm)ƀtypeƀSOURCE=srce $
```

label - label name

du(dm) - specifies the name of the unit and member on which the table is to be built

type - specifies the type of table to be built. A present type must be 1 (see Section 3.4.3.1 for description of Table Type 1)

srce - specifies the location of the table description card images may assume one of two forms:
* card images follow in input stream;
du(dm) card images reside on the specified unit member in card image (CI) format

The table type 1 table description cards are of the following format:

INT = C1,...
IND$_n$ = FC,NV,EXU,EXL,VAL1,VAL2,...
DEP = FC,VAL1,VAL2,...

where:

INT card - contains the integer codes for the interpolation procedures permitted on this table.

C1 - Integer code
0 - no interpolation permitted
1 - linear interpolation

IND$_n$ card - contains the description of the nth independent variable where $1 \leq n \leq 3$.

FC - (format code) the alpha data type code of the nth independent variable.
0 - ordered position from 1 to NV; independent variable values not entered
I - integer
RS - real single precision
RD - real double precision

NV - number of values for the nth independent variable

EXU - integer code for the extrapolation procedure to be used (by interpolation routines) if the independent variable is greater than the largest independent variable
0 - no extrapolation allowed
1 - use the independent value closest to the specified value
2 - extrapolation is linear using the last two independent values

EXL - integer code for the extrapolation procedures to be used if the independent variable is less than the smallest independent variable. See EXU for code value

VAL1 - NV values for the nth independent variable in ascending or descending order. Values are separated by blank or comma and may extend over several card images. If FC=0, values are not included.

DEP card - contains the description and values of the dependent variable and must follow the INDN cards.

FC - (format code) alpha data type of dependent variable
I - integer
RS - real single precision
RD - real double precision

VAL1 - values for the dependent variable separated by commas or blanks. May extend over several card images. The order of dependent variables is such that the first independent variable varies first, the second variable varies second, and the third variable varies third.

END* card - required if source = *.

Examples:

```
LAB  TABLE UNI(DMS),1,SØURCE=* $
INT = 0,1
IND2 = I,2,0,1,5,10
IND1 = RS,3,0,1,1.5,2.0,4.5
DEP = I,3,5,7,8,9
END* $
TABLE UN2(DM1),1,SØURCE=UN5 (DM2) $
```

Restriction: TABLE with SØURCE=* is valid in the Primary Iuput Stream only. TABLE with SØURCE=DU(DM) is valid in the Primary or Secondary Iuput Stream.

3.5.2.4.23 UNLØAD

Purpose: The UNLØAD control statement unloads data units, known to the present ANOPP run, to a sequential library file which has been defined and assigned to the run through the operating system control language.

Format:

$$[\text{label}_d]\text{UNLØAD/sefn/}[\text{dus}_1,\ldots\text{dus}_n]\ \$$$

where dus is of the following form:

$$\text{du}[(\text{dm}_1,\ldots\text{dm}_n)]$$

label - label name

sefn - the sequential file name of the library file to which data units are to be unloaded

du - the name of the data unit to be unloaded

dm - the name of the data member to be unloaded. If a data member is not specified, all of the data members on the data unit are unloaded

If a data unit list is not specified, all data units presently defined in the Unit Directory except XSUNIT and DATA will be unloaded to sefn. The Unit Directory consists of all units which have been LØADed, ATTACHed, or CREATed, and have not been DETACHed or PURGed since the beginning of the ANOPP run.

Examples:

```
UNLØAD/TAPE1/ $
UNLØAD/TAPE1/UNIT1,UNIT2(MEM1) $
```

Restrictions: There may be no duplicate data unit and data member name combinations.

3.5.2.4.24 UPDATE

Purpose: The UPDATE control statement provides a means of building a data unit by using an existing data unit as a basis for modifications or by adding members from various sources with no one data unit as a basis or a combination of both. There are two UPDATE modes, revise and create, depending on the presence of a data unit as a basis for revision. For more detailed information concerning UPDATE, see Section 3.8.

Format:

$$[\text{label}_{\not b}]\text{UPDATE}_{\not b}[\text{ØLDU=du}_1{}_{\not b}]\ \text{NEWU=du}_2\ {}_{\not b}\ [\text{ALL}_{\not b}]\text{SØURCE=}\left\{\begin{matrix}*\\ \text{du}_3(\text{dm}_3)\end{matrix}\right\}_{\not b}\left[\text{LIST=x}[\text{x...}]\right]^{\$}$$

label - label name

du_1 - the name of the data unit to be used as the basis for UPDATE processing. The presence of the OLDU clause indicates a revise mode UPDATE. Member level and record level directives which allow a default of OLDU data unit or imply the OLDU data unit, will use du_1, as the required unit. The omission of the OLDU clause indicates a create mode UPDATE.

du_2 - the name of the new data unit to be built during UPDATE processing.

ALL - this keyword indicates a full update of the basis data unit (OLDU) is to be performed. All data members on the OLDU data unit which are not processed by a member level directive will be copied to the NEWU data unit.

SOURCE clause - the SOURCE clause specifies where the set of UPDATE input directives will be found.
* indicates the directives follow immediately in the Primary Input Stream with the set of UPDATE directives terminated by an END* CS.
$du_3(dm_3)$ indicates the directives are found on the data unit and data member specified

LIST clause - specifies the type of printed output required from the UPDATE processing. The list following the = is a sequence of letters specifying the sections of output desired. The list may be any combination of the following:

E - Directive Echo Section
S - Summary Section
C - CHANGE Members Section
A - ADDR Members Section

Examples:

```
LAB1   UPDATE NEWU=U1,SØURCE=*,LIST=S $
          (directive set)
END* $
UPDATE  ØLDU=U001,NEWU=U002,ALL,SØURCE=DUS(M1),LIST=SEA $
UPDATE  ØLDU U002,NEWU ABC,SØURCE=* $
          (directive set)
END* $
UPDATE  ALL,ØLD=UN1,NEW=UN2,SØURCE=*,LIST=A $
              .
              .
              .
          (update directives)
              .
              .
              .
END*
UPDATE  NEW=UN3,SØURCE=UN4(MEM1) $
```

Restrictions: See UPDATE Description (Section 3.8).

3.5.3 Executive Monitor (XM)

Purpose: The Executive Monitor (XM) module is the single driver for the Executive Management System. XM is also the main FORTRAN program for ANOPP and remains in core at all times. It is in ultimate control of ANOPP from beginning to end and thus directs the execution of other Executive Management System modules to accomplish the tasks required. XM calls into execution five of the Executive Management System execution phases. These phases are the Initialization Phase, the Primary Edit Phase, the Control Statement Processing Phase, the Functional Module Processing Phase, and the Error Processing Phase. The remaining execution phases are called into execution as required during these phases by lower level modules.

Input: Since XM is a driver module to direct execution of various execution phases, there is no direct input.

Output: Since XM is a driver module there is no output. All output from ANOPP is accomplished within the various execution phases.

Functional Description: The functions of XM are as follows:

1. To perform initialization requirements for the Executive Management System, the Data Base Management System, the Dynamic Storage Management System, and the General Utilities (Initialization Phase).

2. To validate the ANOPP execution sequence defined by the Primary Input Stream (Primary Edit Phase).

3. To execute or process the execution sequence to completion (Control Statement Processing Phase).

4. To execute or process Functional Modules (Functional Module Processing Phase).

5. To direct the action to be taken upon encountering a non-fatal error during execution of the Control Statement or Functional Module Processing Phase (Error Processing Phase).

Logical Description: XM calls into execution the Initialization Phase, and the Primary Edit Phase and then iterates between the Control Statement Processing Phase and either the Functional Module Processing Phase or the Error Processing Phase.

At the beginning of ANOPP the driver XM is brought into execution and XM immediately calls XLINK requesting that the module XBS be executed. XBS controls the Initialization

Phase. Initialization requirements for all executive modules are performed according to parameter values specified on the optional ANOPP CS in the Primary Input Stream or default values provided by the ANOPP installation settings.

Upon completion of XBS, XM calls XLINK requesting that XRT be executed. The module XRT controls the Primary Edit Phase. The control statements in the Primary Input Stream following the STARTCS are read, validated, reformatted into an executable form, and written as data member M001. This member resides on the data unit XSUNIT which is reserved for Executive Management System usage. The reformatted control statements residing on M001 are called the Primary CS Set. XRT does not return to XM if during the Initialization Phase or the Primary Edit Phase an error has occurred; XRT will call the Error Termination Phase to abort ANOPP.

Upon completion of the Primary Edit Phase, XM calls XCSP via XLINK. The XCSP module controls the Control Statement (CS) Processing Phase. The Primary CS Set is executed or processed sequentially allowing for flow alteration by certain control statements. A Secondary Input Stream may be called into execution via a CALL CS. Upon the first execution of a particular CALL, the Secondary Input Stream is read, validated, reformatted, and written in executable form as an Mxxx type data member residing on the XSUNIT data unit. The name Mxxx, where xxx is an integer sequentially assigned as required, is assigned to the data member. This data member is called a Secondary CS Set. XCSP continues to execute the Primary and Secondary CS Sets to completion unless a non-fatal error is detected or execution of a Functional Module is requested via an EXECUTE CS. In either case, control is returned to XM for action.

Upon return from XCSP, XM determines the reason for interruption of the CS Processing Phase. The ANOPP error indicator, variable NERR residing in /XCVT/, is interrogated and if errors occurred, the module XMERR is called via XLINK to perform the Error Processing Phase. If errors occurred then the XFM module is called directly by XM to perform the Functional Module (F.M.) Processing Phase.

During the Error Processing Phase, depending on the system parameter JCØN, XMERR will provide the environment for the next entry to XCSP to either continue processing with the next CS or to continue processing with the first PRØCEED CS found in a sequential scan forward from the CS which encountered the error. No control statements are executed during the scan. In particular, a call statement is neither expanded nor processed. If no PRØCEED is found in the search, XMERR provides the environment to continue processing with the ENDCS in the Primary CS Set; this will subsequently invoke the Normal Termination Phase upon the next entry to XCSP.

Upon return to XM from XMERR, the module XCSP is called to proceed with the CS Processing Phase.

During the Functional Module Processing Phase, XFM brings into execution the requested Functional Module (F.M.). Upon completion of the F.M. the integrity of the ANOPP system environment is validated and insured by XFM taking corrective action as required.

Upon return to XFM, XM interrogates the ANOPP error indicator NERR to determine non-fatal error occurrence during the F.M. Processing Phase. If error occurrence is detected then XMERR is called via XLINK to perform the Error Processing Phase. Upon return from XMERR, XM calls XCSP to proceed with the CS Processing Phase.

After once calling XBS and XRT, XM cycles between calling XCSP and XFM or XMERR.

Error Philosophy: No error is detected directly within the driver XM. However, error detection does occur within the execution phases called into execution by XM. If a fatal error, which inhibits recovery with further processing, is detected then ANOPP is abnormally terminated via the Error Termination Phase and there is no return to XM. If a non-fatal error is detected within XBS, the indicator NERR is set. XM does not detect this error return from XBS and allows XRT to be called. XRT upon completion will recognize error occurrence during execution of XBS or itself and will abnormally terminate via the Error Termination Phase. If a non-fatal error is detected within XCSP or XFM then the NERR indicator is set and action is taken upon return to XM.

3.5.4 Execution Phases

The Executive Management System (EM) includes eight execution phases corresponding respectively to the eight tasks given in the Overview Section (Section 3.5.1). These phases, along with the controlling EM Module, are as follows:

1. Initialization Phase (XBS)
2. Primary Edit Phase (XRT)
3. Control Statement Processing Phase (XCSP)
4. Functional Module Processing Phase (XFM)
5. Error Processing Phase (XMERR)
6. Secondary Edit Phase (XCA)
7. Normal Termination Phase (XEN)
8. Error Termination Phase (XXFMSG)

3.5.4.1 Initialization Phase (XBS)

Purpose: The XBS Module controls the Initialization Phase. The ANOPP Title Page is printed and all initialization requirements for DBM, DSM, EM, and the General Utilities are performed.

Input: Input to XBS is the Primary Input Stream which contains the optional ANØPP CS and the required STARTCS CS.

Output:

1. Data Base Structures
   XSUNIT - the data unit created for subsequent usage by the Executive Management System (EM) for scratch data members temporary to ANOPP.
   DATA - the data unit created for subsequent usage by EM in processing the DATA control statements.

2. Common Block Variables
   Required variables in the following common blocks are initialized:
   /XCS/, /XCSFM/, /XCVT/, /XDBMC/, /XDSMC/, /XSPT/.

3. Control Structures
   The following Control Structures are allocated and initialized in Global Storage:
   Alternate Name Table (ANT)
   Data Unit Directory (DUD)

Member Description Blocks Table (MDBT)
User Parameter Table (UPT)
User String Table (UST)

Active Member Directory (AMD)
Data Table Directory (DTD)
Library File Directory (LFD)
Member Directory (MD) work area

Functional Description: The XBS module performs the ANOPP Initialization Phase requirements. The Primary Input Stream is processed through the STARTCS control statement. If the optional ANØPP CS is present, the values specified for the ANOPP system parameters replace the ANOPP system default values for subsequent initialization procedures. If the ANØPP CS is omitted, the ANOPP system default values are used for subsequent initialization procedures.

DSM initialization requirements include setting parameter values in the common block /XDSMC/ and initializing Global Dynamic Storage according to the length required.

DBM initialization requirements include setting parameter values in the common block /XDBMC/, creating the data units DATA and XSUNIT for EM utilization, and allocating required control structures. These control structures are the DUD, DTD, AMD, and LFD. A working storage block for the MD is also allocated. The DUD and DTD contain information which must not be moved to other core locations when DSM consolidations occur; thus the DUD and DTD are the first allocated blocks in GDS.

EM initialization requirements include setting parameter values in the common blocks /XCVT/, /XCS/, /XCSFM/, and /XSPT/. The following control structures are allocated and initialized in GDS: MDBT, ANT, UPT, and UST. The ANT is initialized for zero allocated entries. The others are initialized using installation default values.

There are no additional initialization requirements imposed by the General Utilities.

Logical Description: XBS performs initialization functions for EM and calls four additional modules to perform the remaining Initialization Phase requirements.

Immediately upon entry the module XBSTP is called to print the standard ANOPP title page.

The module XBSIN is then called by XBS to determine the presence or absence of the ANØPP CS in the Primary Input Stream and to initialize parameters according to either the ANØPP CS specification or the installation default values. Detection of the STARTCS in the Primary Input Stream is also accomplished.

The module XBSDSM is called by XBS to initialize Global Dynamic Storage according to the length specified either by the ANØPP CS or by the installation default value.

The module XBSDBM initializes the Data Base Management System Control and data base structures. The Data Unit Directory (DUD) and Data Table Directory (DTD) are insured to be the first blocks allocated in GDS. They are not expandable tables and thus are allocated for fixed numbers of entries which may not be exceeded during ANOPP execution. The AMD, MD, and LFD control structures are allocated according to parameterized default values. The data units, XSUNIT and DATA, are created with corresponding entries made in the DUD.

Upon completion of DSM and DBM initialization, XBS allocates the EM control structures which include the MDBT, ANT, UPT, and UST. An entry in the MDBT is allocated for the Primary CS Set, or M001 member, and the entry is initialized. Alternate names exist only during the Functional Module Processing Phase, thus the ANT is allocated for zero-entries. The UPT and UST are allocated according to installation default values.

XBSTP, XBSIN, XBSDSM, and XBSDBM are the primary modules called by XBS to perform Initialization Phase functions.

Error Philosophy: If an error occurs in allocating or initializing any control or data base structure, then ANOPP is abnormally terminated. Fatal errors encountered while initializing DBM and DSM are processed respectively via the member manager module MMERR and the DSM module DSMERR. Other fatal errors invoke the EM Error Termination Phase for processing.

An error encountered in processing the ANØPP CS is not immediately fatal. The Initialization Phase continues to completion and the ANOPP error indicator, NERR, is set to .TRUE. upon exit from XBS. ANOPP is subsequently terminated upon completion of the Primary Edit Phase.

3.5.4.2 Primary Edit Phase (XRT)

Purpose: The Primary Edit Phase (XRT) module is called by the Executive Monitor (XM) module after the Initialization Phase (XBS) module has completed its task. The Primary Edit Phase examines and validates control statements in the Primary Input Stream and builds a record for each CS on the root member, M001, in a format that is recognized by the Control Statement Processing Phase (XCSP). The Primary Edit Phase allocates a new Mxxx member name in sequential order each time a CALL CS is encountered in the Primary Input Stream beginning with M002. A Member Description Block entry (MDB) is initialized in the MDBT for the Mxxx just allocated. The Primary Edit Phase builds a Uxxx type member each time an UPDATE or a TABLE CS is encountered with a SØURCE=* specification. The SØURCE=* specification indicates that input expected for the particular CS will be found in the input stream, beginning with the card image immediately following the CS and including all images down to but not including the END* CS. Upon completion of its task, the Primary Edit Phase will terminate ANOPP processing if an error was detected in the Initialization Phase prior to XRT entry or in the Primary Edit Phase, or if the user option to terminate after Edit Phase is set. Otherwise, the Primary Edit Phase will return control to the Executive Monitor (XM).

Input: The primary input to the Primary Edit Phase (XRT) is the Primary Input Stream which begins with the control statement following the STARTCS statement and ends with the ENDCS statement. Other pertinent input is described below; particular input required for CS processing is not necessary for understanding and is not included.

1. Data Base Structures
   XSUNIT - the ANOPP system scratch unit, XSUNIT, contains no members.
   DATA - the DATA unit contains no members.

2. Common Block Variables
   /XCS/
   NØGØ - Primary-Edit-only indicator is set to .TRUE. if processing is to be stopped when the Primary Edit Phase is complete. When the indicator is set to .FALSE. the Primary Edit Phase returns to XM upon completion.

   /XCVT/
   NERR - Executive System Logical Error indicator is set to .TRUE. if an error was detected in the Initialization Phase (XBS). In case of

such an error, XRT edits and builds CS records but suspends writing of those records to the M001 member. At completion of the Primary Edit Phase processing will be terminated.

3. Control Structures

MDBT - the Member Description Block Table resides in GDS and is a system table type 1. An entry (MDB) in the MDBT is initialized for the M001 root member. Initial settings in the MDB indicate that the M001 root member has yet to be constructed.

Output:

1. Data Base Structures

M001 - The primary output of the Primary Edit Phase is the Primary CS Set residing as the M001 root member on XSUNIT in a format that is recognized by the Control Statement Processing Phase. M001 contains a variable length control statement (CS) record for each complete CS edited in the Primary Input Stream and a Label Record that provides a cross-reference to each labeled CS on the M001 member. If an error is detected, writing to the M001 member is suppressed, but editing continues to the last CS in the Primary Input Stream.

Uxxx - A Uxxx type data member in card image (CI) format resides on XSUNIT unit for each UPDATE CS or TABLE CS with SØURCE=* specification encountered in the Primary Input Stream. A Uxxx type data member resides on DATA unit for each DATA CS encountered in the Primary Input Stream. The Uxxx contains the card image input to the CS which will be utilized in subsequent execution of the CS.

2. Common Block Variables

/XCS/

MEMCUR - The current member in execution is defined by MEMCUR as M001.

/XCVT/

NERR - On exit from the Primary Edit Phase (XRT) the executive system logical error indicator is always set to the no error condition of .FALSE.

3. Control Structures

MDBT - The Member Description Block entry for the (MDB) M001 root member is in executable format. Also, an MDB entry is initialized for each Mxxx type member name assigned as a result of a CALL CS successfully edited in the Primary Edit Phase. Initial values in the MDB indicate that the Mxxx member has yet to be constructed.

Functional Description: The XRT module performs all operations necessary in constructing the M001 root member. The Primary Input Stream is processed beginning with the control statement that follows the STARTCS control statement. Processing terminates when the ENDCS control statement is encountered. An unformatted variable length control statement record is built for each complete control statement image in the Primary Input Stream. The control statement records are written on the M001 root member unless an error is detected. If an error occurs, writing on the M001 member is suppressed, but editing and building CS Records continues until the ENDCS control statement is encountered. A Label Record is built identifying the number of every CS Record where a label is present and giving the label name. The Label Record is written as the last record on the M001 root member.

As the Primary Input Stream is processed and the control statement records are built, the control statement images are echoed if the system parameter JECHØ is set to .TRUE. or an error is detected in building the CS Record. If an error is detected in building the M001 member, the system parameter JECHØ is automatically set so that subsequent CS images will be echoed, and writing on the M001 member is inhibited. A control statement is considered in error under any of the following circumstances:

1. The maximum number of continuation cards allowed per CS is exceeded.
2. An unrecognizable field is detected on the CS image.
3. The form of a label field is invalid or a duplicate label is detected.
4. CS name is invalid to the Primary CS Set.
5. Syntax of the CS is invalid for the corresponding CS name.

If a CS image exceeds the maximum allowable card images for a valid CS, the control statement is arbitrarily terminated at the end of the last allowable image and processing continues as it would for a valid CS Record. The image immediately following this CS in the Primary Input Stream will be read and processed as the next CS image.

A syntax check is performed on each Control Statement Record to insure that the format of the CS meets the requirements of the particular CS name.

All references to CS labels are verified. CS names valid to the Primary CS Set require special processing during the Primary Edit Phase. All label references on the IF and GØTØ control statements are entered in the Label Reference Table maintained by the XRT module. When all control statements have been processed the table is used to validate that all labels which have been referenced are present in control statements in the Primary Input Stream.

A comment CS is one in which the end-of-data character ($) is the first non-blank character on the CS image. A comment control statement is included as a CS Record with CØNTINUE substituted as the CS name.

A DATA CS is processed completely in the Primary Edit Phase. CØNTINUE is substituted as the CS name in the CS Record and the images following the DATA CS in the Primary Input Stream down to the END* CS are written as a data member on the system unit DATA in card image (CI) format. The END* CS which indicates the end of input for the DATA member is not included on the member. The data member name is specified on the DATA CS.

For each CALL CS encountered in the Primary Input Stream, an Mxxx data member name is assigned and an MDB entry is made in the Member Description Blocks Table with initial settings. The initial setting indicates the Mxxx member has not been constructed and does not exist on XSUNIT. Mxxx data member names are assigned sequentially where the xxx portion of the name is Hollerith digits 002-999. The Mxxx member will be constructed during the Secondary Edit Phase upon the first execution of the CALL CS in the CS Processing Phase.

The UPDATE and TABLE control statements require special processing when SØURCE=* is specified. The SØURCE=* specification indicates that input for the UPDATE or TABLE follows the CS in the Primary Input Stream and includes all images until an END* CS is detected. These card images are saved on a Uxxx member with one card image per record in CI format. A Uxxx type data member is built on the system data unit XSUNIT each time such an UPDATE or TABLE CS is detected. Uxxx member names are assigned sequentially with the xxx portion of the name being Hollerith digits 001-999.

If any CS requiring special handling of the card images immediately following in the input stream (DATA, TABLE, or UPDATE) is in error, then the images in the input stream will be skipped through the END* CS and will not be processed as described above.

The M001, root member, is considered in error if a control statement in the Primary Input Stream is found to be in error or if an unsatisfied label reference is detected. When the M001 member is considered in error, writing to all data units (XSUNIT and DATA) is suspended as indicated above.

Detection of the ENDCS control statement indicates the end of the Primary Input Stream. If the end of the input file is detected before the ENDCS statement is encountered, and ENDCS statement is simulated for complete recovery. The XRT module makes a normal return if all of the following conditions are met:

1. Primary Input Stream processing is complete (an ENDCS Control statement is detected or simulated)
2. The M001 root member is error free (no error was detected on any edited control statement)
3. The user option to terminate after Primary Edit Phase (NØGØ) is not set.
4. The Initialization Phase was error free.

If all of these conditions are not met, the XRT module aborts upon completion of Primary Iuput Stream processing.

Logical Description: Immediately upon entry to the XRT module, the XRTI module is called to allocate and initialize expandable Local Dynamic Storage blocks required to build the Control Statement Record, the Label Record, and the Label Reference Table, and to initialize appropriate variables in the /XRØØT/ common block. The CS Record block is allocated to build the maximum length CS Record. The Label Record and Label Reference blocks are allocated for an arbitrary number of label entries.

The XRT module then opens the M001 root member for write via scratch access and begins to process the Primary Input Stream. The following process is iterated for each control statement read until the ENDCS statement is encountered:

A. The XCR module is called to crack each image read from the Primary Input Stream. XCR builds a table from each image that includes every field detected on the CS image preceded by an integer type code field identifying the field as one of the ANOPP field types.

B. If the maximum number of images per CS is read and the end-of-data character ($) is not detected the XRTTC module is called to simulate a complete control statement as if it is complete.

C. The XRTBAD module is called if unrecognizable fields are detected on the current control statement. XRTBAD prints each unrecognizable field found on CS.

D. The XRTBCS module is then called to build a valid control statement record from the cracked table produced by XCR. XRTBCS strips off the first, and possibly the second, field in the cracked field table looking for a valid label name field, if present, and a CS name valid to the Primary CS Set. The label name, if present, and the CS name are entered in the CS Record. Upon exit from XRTBCS, the CS Record is complete and is read to be written on the M001 root member.

E. If a valid label field was detected by XRTBCS then the XRT module calls XRTBLR module to add an entry to the Label Record. Each entry in the Label Record identifies the number of the CS record and the label name.

F. If the CS is still error free after the syntax check, XRT calls XRTLRF module to pick up label references from IF and GØTØ control statements. Label references are entered in a single-word entry in the Label Reference Table.

G. XRT then echoes the CS image if the user print option JECHØ is set or if an error was detected in the current CS or a previous CS.

H. The XRTCSS module is then called to process the special CS names, DATA, CALL, UPDATE, and TABLE. A DATA control statement is processed completely, substituting CØNTINUE as the CS name in the CS Record Preface, and writing data input on the specified data member on the system unit named DATA in card image (CI) format. For a CALL CS an Mxxx data member name is assigned and the member name is entered in the CALL CS Record Preface. A corresponding MDB entry is also made and initialized in the MDBT. Special processing is required for the UPDATE and TABLE statements if SØURCE=* is specified. For these control statements a Uxxx member name is assigned, input for the CS is written on the member in CI format, and the Uxxx member name is entered in the UPDATE or TABLE CS Record Preface. Upon exit from the XRTCSS module all special CS processing is complete.

I. Current CS processing is now complete and the CS Record is written on the M001 root member if the Primary Input Stream has been error free and there were no Initialization Phase Errors.

Primary Input Stream processing is complete when an ENDCS control statement is edited and the corresponding CS record placed on M001. Then the XRTLSA module is called to validate that all label references (found in the Label Reference Table) have been satisfied. If all label references are satisfied then the XRT module writes the Label Record on the M001 root member as the last record and enters the label record length in the member description block entry in the MDBT for the M001 root member.

The M001 root member is now complete and is closed by the XRT module. XRT then defines M001 as the Mxxx member now in execution by setting the output variable MEMCUR. The XRTRS module is then called to free the Local Dynamic Storage blocks used for building the CS Record, the Label Record, and the Label Reference Table and to release Local Dynamic Storage.

If an error was detected in the Initialization Phase or while building the M001 member, or if the NØGØ parameter is set to .TRUE., then XRT aborts. Otherwise, a normal return is made to XM.

Error Philosophy: The Primary Edit Phase aborts via XXFMSG fatal message writer if an error is detected while opening the M001 root member or if there is insufficient Local Dynamic Storage to expand any of the XRT expandable LDS blocks.

A missing ENDCS control statement in the Primary Edit Phase does not make further processing impossible, so in such a case, an ENDCS statement is simulated for complete recovery.

Errors detected in the Primary Input Stream while building the M001 member are not immediately fatal. Errors detected in the Primary Edit Phase will result in error messages printed before the appropriate CS image is echoed, will inhibit writing to the M001 member, and will result in an error flag setting for the M001 member. Editing and building CS Records will continue until the Primary Input Stream has been completely processed, and XRT will then abort the Primary Edit Phase.

3.5.4.3 Control Statement Processing Phase (XCSP)

Purpose: The XCSP module controls the Control Statement (CS) Processing Phase. The Primary Input Stream is executed from beginning to end allowing for execution of optionally supplied Secondary Input Stream(s). This phase is temporarily interrupted with return to the driver XM whenever either a non-fatal error is encountered or execution of Functional Module is requested.

Input: The primary input to XCSP is discussed below. Additional input, although required for particular CS processing, is not necessary for understanding and is not included.

1. Data Base Structures

   XSUNIT - The unit XSUNIT contains Mxxx members and Uxxx members where xxx is display code of an integer 001-999. M001 is the Primary CS Set or root member and is always present. There is an Mxxx member where xxx is greater than 001 for each Secondary CS Set which has been brought into execution at least once via a CALL CS. There is no limit to the number of Mxxx members which may be in completed execution or suspended execution but there is one and only one Mxxx which is in current execution. A Secondary CS Set is in completed execution if it has been brought into execution via a CALL CS and execution has been completed. A Primary or Secondary CS Set is in suspended execution if its execution has been temporarily interrupted by a Secondary CS Set brought into execution via a CALL CS. A Primary or Secondary CS Set is in current execution if it contains the next CS to be executed. Any Mxxx member in current or suspended execution contains at least one CS record to be executed. For M001 this is the ENDCS CS and for Mxxx it is the RETURN CS. A Uxxx member exists for each TABLE CS and for each UPDATE CS with SØURCE=* specification in the Primary CS Set. All Mxxx and Uxxx members are closed.

DATA - The Unit DATA contains a member corresponding to each DATA CS encountered in the Primary Input Stream during the Primary Edit Phase.

2. Common Block Variables

/XCS/

MEMCUR - name of the Mxxx member in current execution

MXMDB - IDX of the Member Description Block Table (MDBT) residing in GDS

MNAME,MCUR,MCALL,MRL,MLL - position parameters for a Member Description Block (MDB) which is an entry in the MDBT

/XCSFM/

LANT - IDX of the Alternate Names Table (ANT) residing in GDS

LUPT - IDX of the User Parameter Table (UPT) residing in GDS

LUST - IDX of the User String Table (UST) residing in GDS

3. Control Structures

MDBT - the Member Description Block Table (MDBT) resides in GDS and is a System Table Type 1 which contains a Member Description Block (MDB) entry for each Mxxx name assigned.

ANT - the Alternate Names Table (ANT) is a System Table Type 1 residing in GDS, which defines the active set of alternate names. It always has zero entries on entry. Alternate names provide only an interface capability between the F.M. and the CS Stream being executed and thus exist only during the F.M. Processing Phase.

UPT/UST - the User Parameter Table (UPT) is a System Table Type 1 and the User String Table (UST) is a System Table Type 2; both reside in GDS. The UPT and UST in combination define all user parameters. There is an entry in the UPT, and UST if required, for each user parameter currently defined.

4. Initial Entry

For the initial entry to XCSP the following environment exists:

a. XSUNIT contains the M001 member and Uxxx members may or may not exist.

b. DATA may or may not contain members.

c. MEMCUR contains the name M001.

d. The MDBT contains an executed MDB for M001 with the MNAME position = M001, MCUR position = 0, MCALL position = blanks, MRL position $>$ 0, and MLL position $>$ 0.

e. The UPT, UST, and ANT contain zero entries. All data members are closed.

<u>Output</u>:

1. Data Base Structures

All members on the units XSUNIT and DATA are closed.

2. Common Block Variables

The primary output from XCSP upon return to the XM driver is definition of the reason for CS Processing Phase interruption and definition of the specific CS in a CS Set with which processing will continue upon resumption of the CS Processing Phase. This information is provided through common block variables.

/XCVT/

NERR - the logical ANOPP error indicator. It is set to .TRUE. if a non-fatal error occurred during processing a CS thus causing the interruption; otherwise it is .FALSE.

/XCS/

REQ - if an EXECUTE CS precipitated the interruption of XCSP then REQ contains the integer corresponding to the specific Functional Module (F.M.) requested. Integer and F.M. correspondence is determined upon F.M. installation.

MEMCUR - the name of the current Mxxx member being executed.

MXMDB - the IDX of the MDBT in GDS. The current CS record pointer (MCUR) in the MDB entry for the current Mxxx member is set to the CS record resulting in the interruption.

Functional Description: The CS Processing Phase begins with execution of the first CS in the Primary CS Set which is contained on the M001 member on XSUNIT. Execution of subsequent control statements in the Primary CS Set is sequential through the final CS which is the ENDCS. The sequential flow, however, may be altered by special control statements which transfer execution to a labelled CS within the Primary CS Set. One such CS is the GØTØ. After such an alteration, sequential execution is resumed, thus the ENDCS is eventually executed. Execution of the ENDCS invokes the Normal Termination Phase which terminates ANOPP.

Execution of the M001 member is temporarily suspended upon execution of a CALL CS. The CALL CS requests that a Secondary Input Stream be brought into execution and completed before continuing execution of the current M001 member.

Upon the first execution of a CALL CS, the corresponding Secondary CS Set has not yet been constructed and does not exist on XSUNIT as a Mxxx member. However, during the Primary Edit Phase when M001 was constructed, an Mxxx name was assigned for each encountered CALL CS and a corresponding MDB entry in the MDBT was allocated. The MDB entry has initialized settings which indicate the corresponding CALL CS has not previously been executed thus the Mxxx member does not exist. Upon the first execution of a particular CALL CS, the Secondary Edit Phase (XCA) is invoked to validate the specified Secondary Input Stream and to construct a corresponding Secondary CS Set residing on XSUNIT as the Mxxx member previously assigned. The Mxxx member has the same format as the root member, M001, and it contains the set of control statements which compose the Secondary Input Stream in executable form. The last CS record in the Mxxx member is a RETURN CS simulated during the Secondary Edit Phase to allow eventual return of control to M001. Upon completion of the Secondary Edit Phase, the M001 member is put in suspended execution by transferring execution control to the new Mxxx member.

Upon a subsequent execution of a particular CALL CS, the specified Secondary Input Stream does exist in executable form as the Mxxx member previously constructed during the Secondary Edit Phase. Thus, the Secondary Edit Phase is not invoked and the M001 member is put in suspended execution immediately by transferring execution control to the corresponding Mxxx member.

When execution control is transferred to a Secondary CS Set, execution begins with the first CS record and continues sequentially, until the final CS, the RETURN, is encountered. The sequential flow may be altered, as in the Primary CS Set. Execution of the RETURN indicates the Secondary CS Set has been completed and control is transferred back to the Primary CS Set. Execution resumes with the CS immediately following the CALL CS which suspended execution by the Primary CS Set.

During execution of a Secondary CS Set, a CALL CS may also be encountered. The currently executing Secondary CS Set is put in suspended execution, the Secondary Edit Phase is invoked upon the first execution of the particular CALL and execution control is transferred to the specified Mxxx member, or Secondary CS Set. The process invoked by encountering a CALL CS in a Secondary CS Set is identical to the process invoked by encountering a CALL CS in the Primary CS Set.

There is no limit to the number of Mxxx members which may be in suspended execution. Upon completion of the "called" Secondary CS Set, the "calling" CS Set is brought back into current execution and execution resumes with the CS immediately following the CALL CS which invoked the suspended execution. Eventually as the suspended Secondary CS Sets are one by one brought back into current execution and completed, the Primary CS Set is brought back into current execution. The last CS record in the Primary CS Set is the ENDCS which when executed invokes the Normal Termination Phase. The Normal Termination Phase terminates ANOPP with no return to XCSP. Thus, ANOPP is terminated by invoking the Normal Termination Phase upon execution of the ENDCS during the CS Processing Phase.

The CS Processing Phase is temporarily interrupted by two conditions. The first is the request for a Functional Module (F.M.) to be executed via the EXECUTE CS. The second is a non-fatal error occurrence while processing a CS record.

The EXECUTE CS is processed by XCSP as follows. The Alternate Names Table (ANT) is constructed according to the alternate name specifications on the EXECUTE CS. The F.M. and ANOPP executive modules will utilize the ANT during the F.M. Processing Phase which follows XCSP interruption and return to the driver XM. A set of alternate names defined by the ANT is valid only during the F.M. Processing Phase. All entries in the ANT are deleted upon completion of that Phase. Thus, the ANT always has zero entries upon entry to XCSP. The name of the F.M. to be executed has been pre-processed during the Primary or Secondary Edit Phase; and the EXECUTE CS record upon execution contains an integer which corresponds to the F.M. name specified on the CS card image in the Primary or Secondary Input Stream. The correspondence between a particular integer and a particular F.M. is unique and was assigned when the F.M. was installed into ANOPP. This integer which is sufficient for the F.M. Processing Phase to determine, load, and execute the proper F.M. is placed in the XCSP output variable REQ (/XCS/ common block).

If a non-fatal error occurs during the processing of any CS, the ANOPP error indicator NERR is set to a .TRUE. value. Whether or not that CS processing was completed or partially completed depends on the particular CS.

Upon XCSP interruption and return to the driver XM, the existing environment is defined sufficiently to allow for resuming the CS Processing Phase by a subsequent entry to XCSP. All data members utilized are closed.

All CS records are processed by the XCSP modules according to individual requirements. The card images of the CS are printed as the CS is executed depending on the system parameter JECHØ value.

Logical Description: In the following logical description of XCSP, there is no attempt to discriminate between the initial entry or a subsequent entry to XCSP. Although the initial entry to XCSP always begins processing with the first CS record in M001 and a subsequent entry resumes processing with a CS record in any Mxxx member, there is no logical differentiation required internal to the XCSP module. The first entry to XCSP is logically identical to any subsequent array. The environment which is defined by the input to XCSP dictates either the beginning of CS processing or a resumption of CS processing and both environments are processed identically.

On entry, XCSP calls the XCSPM module to initialize the environment to resume processing with the next CS record in the currently executing Mxxx member. Local Dynamic Storage (LDS) is initialized. The Mxxx member, defined by MEMCUR, with which execution is to begin is opened to read. In the MDBT, the MDB corresponding to Mxxx contains the length of the largest CS record at MRL position and the length of the label record at MLL position. This information is used to allocate in LDS a block large enough for any CS record on Mxxx and a block sufficiently large for the label record. The label record on Mxxx is then read into the LDS label block to define all valid label names and the corresponding CS records for this Mxxx member. The MCUR entry of the MDB contains the position of the last CS record executed. The Mxxx member is positioned so that the next record read will be the CS record with which execution is to resume.

Upon return from the XCSPM module, XCSP enters an indefinite iteration of processing the next CS in the current Mxxx member. The current record pointer is incremented by one to define the position of the CS record to be executed. Mxxx is positioned to the CS record to be executed and it is read into the CS record block in LDS. The card image of this CS is then printed if the system parameter JECHØ is set accordingly. The name of the CS is identified and the appropriate module is called to process the CS. A list of the valid CS names for processing by XCSP along with the name of the module called by XCSP to execute the CS is given in Table 3.

All the modules which process a CS have the same input available and must satisfy the same output requirements with respect to XCSP interface. The input uniformly required is the CS record which fully defines the act to be performed. Additional input is specific to the particular CS. The output environment which must be provided upon return to XCSP must allow XCSP to continue its iterative processing. Specifically, the output must include the following:

1. MEMCUR defines the currently executing Mxxx member.

2. The MCUR entry in the MDB corresponding to the current Mxxx must contain the position of the next CS record to be executed minus 1. In most cases where the sequential flow is unaltered, the value will be unchanged from the input value and will reference the CS record just executed. In other cases where a control statement alters the sequential flow, the value must be reset such that it points to the CS immediately before the next CS to be executed. This setting always allows XCSP to increment the current record position by 1 in continuing the iterative loop or upon a subsequent entry of XCSP when the iterative loop is again begun.

3. The label block in LDS contains the label record of the current Mxxx.

4. The CS record block in LDS must be large enough for any CS record on the current Mxxx. Usually the CS record for the CS just completed has remained unaltered in the block but this is not required. The size of the LDS block, but not necessarily the contents, must be insured.

5. Mxxx is opened to read and is positioned such that the next record read will correspond to the next CS to be processed.

6. The current Mxxx member is the only member open for any reason.

Most of the CS processing modules perform functions which are independent of XCSP interface requirements. The basic input to these modules is the CS record which fully defines the action to be performed. Other input for particular CS processing modules may be required but is not relevant to nor dependent upon XCSP interfaces. Upon entry to these modules the XCSP output requirements are automatically satisfied and remain unaltered upon return to XCSP. No change has occurred to alter the sequential execution

Control Statement Name	ANOPP Executing Module
ARCHIVE	XAR
ATTACH	XAT
CALL	XCA
CØNTINUE	XCØ
CREATE	XCT
DETACH	XDT
DRØP	XDR
ENDCS	XEN
EXECUTE	XEX
GØTØ	XGØ
IF	XIF
LØAD	XLD
PARAM	XPA
PRØCEED	XPR
PURGE	XPU
RETURN	XRE
SETSYS	XSS
TABLE	XTB
UNLØAD	XUN
UPDATE	XUP

Table 3. Valid ANOPP Control Statement Names Processed by XCSP.

thus CS processing always continues with the CS record immediately following the CS just completed. The modules not of this type are those which process the CALL, ENDCS, EXECUTE, GØTØ, IF, and RETURN control statements.

Processing of the EXECUTE CS by the XEX module does affect XCSP execution and has additional output requirements. The required output environment is also present on entry to XEX as in the majority of CS processing modules. The XEX module additionally provides upon return to XCSP the integer corresponding to the F.M. to be executed via the common block /XCS/ variable REQ. XEX also builds the ANT in GDS to include alternate names specified on the EXECUTE CS. Upon entry to XEX, the ANT always has zero allocated entries and upon exit the ANT has the exact number of alternate names specified on the EXECUTE CS which is zero or greater. Upon return to XCSP, a local variable is set to indicate that XCSP iterative processing is to be interrupted due to a F.M. execution request.

The GØTØ CS and the IF CS usually change the execution sequence upon return to XCSP and thus the modules XGØ and XIF must insure the output requirements for XCSP interface are satisfied. The GØTØ CS will always transfer execution control to a CS record in the current Mxxx member whereas the IF CS will do so conditionally. The CS to which transfer is made is usually not but may be the CS immediately following the GØTØ or IF statement. Input to XGØ and XIF include the CS record and the label record. The position of the next CS to be executed is determined from the label record. The Mxxx member is positioned to this CS record and MCUR is set to that position minus 1. Other output requirements are satisfied upon entry with no need for alteration.

The ENDCS indicates the normal completion of the Primary CS Set and upon execution the normal termination phase is invoked. The XEN module closes M001 and terminates ANOPP without return to XCSP. Thus XEN has no output requirements.

The CALL CS requests that the current Mxxx member be suspended and a specified Secondary Input Stream be executed; thus, the processing modu'e XCA must insure the output requirements upon return to XCSP are satisfied. Upon entry to XCA, the name of the Mxxx to be executed is contained in the CALL CS record. The MDB entry in the MDBT corresponding to this Mxxx is interrogated to determine if this CALL has been executed previously and the executable form of the Secondary Input Stream has been constructed and is available for usage or if this CALL has not been executed previously and the executable form is not available on Mxxx for usage. If either MRL or MLL entries in the MDB are zero, then Mxxx has not yet been constructed; if either is non-zero then Mxxx has been constructed. If construction has not yet occurred, then XCA invokes the Secondary Edit Phase to validate the Secondary Input Stream and construct the executable form, or Secondary CS Set, as the Mxxx member on XSUNIT. See Section 3.5.4.6 for a full description of the Secondary Edit Phase. If construction has previously occurred, then the Secondary Edit Phase is bypassed.

XCA calls the module XCATRA to transfer execution to the Mxxx member containing the requested Secondary CS Set. The current Mxxx member is closed and the LDS blocks for the CS record and label record are freed. MEMCUR is set to the new Mxxx member to be executed. MCUR in the MDB corresponding to the new Mxxx member is set to zero (0) indicating the next CS to be executed is CS record number one (1). XCATRA then calls XCSPM to perform initialization functions identical to those performed upon entry to XCSP. XCSPM opens MEMCUR to read, allocates an LDS block sufficient for the largest CS record on MEMCUR, allocates an LDS block for the MEMCUR label record and reads the label record into the block, and positions MEMCUR to the next CS record to be executed which is the first CS record. Upon completion of the XCATRA module the currently executing Mxxx member upon entry to XCSP has been suspended and the specified Mxxx has been brought into current execution. The output requirements for return to XCSP are satisfied.

Upon execution of a RETURN CS the module XRE is called by XCSP to reverse the process performed by XCA and return control to the "calling" or suspended Mxxx. The "called" member (MEMCUR upon entry to XRE) is closed, the LDS blocks are freed and LDS is released. The name of the "calling" Mxxx member, defined by the name in the MCALL entry in the MDB of the "called" Mxxx, is retrieved and placed in MEMCUR, thus, bringing back into execution the suspended Mxxx. The module XCSPM is then called to complete the output requirements. XCSPM opens MEMCUR to read, allocates an LDS block sufficient for the largest CS record on MEMCUR, allocates an LDS block for the label record of MEMCUR and reads the label record into the block, and positions MEMCUR to the CS record following the CALL which invoked the suspension. Upon completion of XCSPM the return of control is complete and the output requirements for XRE return to XCSP are satisfied.

When the current CS has been completely processed by the appropriate module and return is made to XCSP, the iterative processing continues by processing the next CS. The iterative processing of CS records continues either until the ENDCS is executed to terminate ANOPP normally or until one of two conditions is encountered. The first condition is the occurrence of a non-fatal error within a CS processing module. Upon return from a CS processing module within which an error occurred, the ANOPP logical error indicator, NERR, is set to .TRUE.; if no error occurred it is .FALSE. . The second condition is the execution of an EXECUTE CS. Upon return from the XEX module, XCSP sets the local variable ISTAT to 1 (one) indicating a F.M. execution request. If either NERR or ISTAT is set to .TRUE. or 1 (one) respectively, then the iterative CS processing is interrupted and XCSP prepares for return to XM. The LDS blocks are freed and the current Mxxx (MEMCUR) is closed. XCSP then returns to XM.

Error Philosophy: There are two types of errors which may occur within XCSP processing, fatal and non-fatal errors.

A fatal error is the detection of a condition which inhibits further XCSP processing. Fatal errors include: a) conditions which should not exist logically within ANOPP, such as the XSUNIT does not exist when an attempt to open an Mxxx member is performed, and b) conditions which prevent further processing to be productive, such as insufficient LDS for required XCSP block allocations. Fatal errors may occur within any module called by XCSP and ANOPP and are abnormally terminated immediately. Most XCSP called modules abort via the EM auxiliary module XXFMSG which invokes the Error Termination Phase (see Section 3.5.4.8 for full description). However, the modules which process the Data Base Management control statements generally utilize the MMERR and TMERR auxiliary modules. Fatal Member Manager errors occurring during processing of Member Manager control statements (ARCHIVE, ATTACH, CREATE, DETACH, DRØP, LØAD, PURGE, and UNLØAD) abort via MMERR. Fatal errors occurring during processing of Table Manager control statements abort via TMERR.

A non-fatal error is the detection of an abnormal condition during CS processing which does not inhibit further productive XCSP processing. Generally, these are user errors resulting from invalid Primary or Secondary Input Streams such as a non existent unit or member specified as a Table Input Stream. The CS processing module detecting the error prints an informative message via the EM auxiliary XXNMSG, the MM auxiliary MMERR, the TM auxiliary TMERR, and the auxiliaries XLDERR and XUNERR. NERR is set to .TRUE. before returning to XCSP. Before attempting to continue CS processing, XCSP will detect the NERR setting and return to the driver XM for error processing.

3.5.4.4 Functional Module Processing Phase (XFM)

Purpose: The XFM module controls the Functional Module (F.M.) Processing Phase. The F.M. specified on the EXECUTE CS which interrupted the CS Processing Phase is brought into execution. Upon completion of the F.M. the integrity of the ANOPP system environment is validated and insured before return to the driver XM.

Input:

1. Data Base Structures

   All data members are closed.

2. Common Block Variables

   /XCS/

   REQ - the integer corresponding to the F.M. to be executed. The correspondence is determined when a F.M. is installed into ANOPP. For each valid F.M. name which may appear on an EXECUTE CS there is a unique integer in the range (NXLEV1+1) through (NXLEV1+NFM) inclusive. The integer corresponding to the requested F.M. was placed in REQ during the CS Processing Phase when the EXECUTE was encountered.

   NXLEV1 - the number of executive modules which are called directly by XM and are loaded at segmentation level 1. These include XBS, XRT, XCSP, and XMERR. The F.M. integer assignments begin with NXLEV1+1.

   NFM - number of F.M. installed. This includes the F.M. names which are available for F.M. testing before permanent installation. These names are FM1, FM2, FM3, FM4, FM5.

   /XCSFM/

   LANT - the IDX of the Alternate Names Table in GDS.

3. Control Structures

   ANT - the Alternate Names Table resides in GDS and is a System Table Type 1 which contains an entry for each alternate name specified on the EXECUTE CS.

Output:

1. Data Base Structures

   All data members are closed.

2. Common Block Variables

   /XCS/

   REQ - the value is zero indicating the F.M. Processing Phase is complete.

   /XCVT/

   NERR - the ANOPP logical error indicator is set to .TRUE. if an error occurred within the F.M. or during the post F.M. "cleanup" procedures which insured the system integrity. If no error occurred, NERR is .FALSE. .

3. Control Structures

ANT - the Alternate Names Table in GDS contains no entries. It is initialized to zero allocated and zero current entries.

Functional Description: The F.M. which corresponds to the integer REQ is loaded and brought into execution. Upon completion of the F.M. the integrity of the ANOPP system environment is validated and insured.

The functions required to validate and insure the integrity of the ANOPP system environment upon completion of a F.M. are called cleanup procedures. Cleanup procedures validate conditions and perform corrective action if the condition is unsatisfied. The conditions validated and the corrective action taken are described below:

1. Condition: LDS has been released
   Corrective Action if condition unsatisfied: LDS is released

2. Condition: All user consolidation locks on LDS and GDS (excluding the master lock on GDS) are released.
   Corrective action if condition unsatisfied: release user locks.

3. Condition: All data members are closed.
   Corrective action if condition unsatisfied: logically close any data member which is open. A logical close of a member is performed by deleting the member from all MM tables indicating activity on the member. If the member was open to write (direct or scratch) the newly written complete or partial member is lost as if the write had never occurred.

4. Condition: all data tables are closed.
   Corrective action if condition unsatisfied: any opened data table will be logically closed and released from core residence. If the table was opened to alter, the altered table is not written to the member as in a normal close; thus, the altered table is lost for subsequent retrieval.

If any of the conditions is unsatisfied, thus resulting in corrective action, the F.M. Processing Phase is considered to be in error and NERR is set to .TRUE. .

The alternate names defined on the corresponding EXECUTE CS are valid only during execution of a F.M., thus the ANT is initialized to zero allocated entries and zero current entries to indicate a null set of names.

Upon completion of the cleanup procedures and ANT initialization, the F.M. Processing Phase is terminated and return is made to the driver XM.

Logical Description: Upon entry to XFM, REQ is validated to insure the integer corresponds to a F.M. installed in the ANOPP system. REQ must satisfy the following conditions:

$$\text{NXLEV1} < \text{REQ} \leq \text{NXLEV1} + \text{NFM}$$

The module XLINK is then called to load and execute the corresponding F.M.

XLINK contains the one-to-one correspondence of integers and F.M. names and calls the appropriate F.M.. Upon completion of the F.M., the XLINK modules returns to XFM.

XFM continues by performing the cleanup procedures via calls to the modules XFMDSM, XFMMM, and XFMTM.

XFMDSM validates that LDS has been released and that all user consolidation locks on LDS and GDS have been released. If necessary LDS is released, all user locks on LDS and GDS are released, and the indicator NERR is set to .TRUE. .

XFMMM validates that all data members are closed. If a data member is found to be in an open state then the member is removed from the MM tables AMD, MCB and NERR is set to .TRUE. . If the member had been opened to write (direct or scratch) the member is closed as if the open had never occurred; thus, the newly written member is unavailable on a subsequent open.

XFMTM validates that all data tables are closed. If a data table is found to be in an open state, it is removed from GDS core residence and NERR is set to .TRUE.. If the table had been opened for alteration, the table is closed as if it had never been opened; thus, the altered table is not written to the data unit for subsequent retrieval.

Upon completion of the cleanup procedures the module XFMANT is called to zero the ANT. The GDS block containing the ANT is freed and a new GDS block allocated for zero entries in the ANT is requested. The ANT is intialized to zero allocated and zero current entries.

XFM is completed and returns to XLINK thereby returning to the driver XM.

Error Philosophy: If an error has occurred during execution of a F.M., then the ANOPP error indicator NERR has been set to .TRUE. before return from the F.M. to XFM via XLINK.

Regardless of the error return status from the F.M., the cleanup procedures are executed and if a corrective action is required NERR is set to .TRUE. .

Upon return from XFM to XM if NERR is set to .TRUE. then XM determines action to be taken.

3.5.4.5 Error Processing Phase (XMERR)

Purpose: The module XMERR controls the Error Processing Phase. A non-fatal error was encountered during processing of a control statement in either the CS Processing Phase or the F.M. Processing Phase. Depending on the value of the system parameter JCØN, either: 1) the CS sequence is searched sequentially forward for either a PRØCEED CS or the ENDCS CS whichever is encountered first; or 2) no action is taken allowing the CS Processing Phase to resume with the next CS following the CS in error.

Input:

1. Data Base Structures

   XSUNIT - the unit XSUNIT contains all Mxxx members constructed by the Primary and Secondary Edit Phases. All Mxxx members are closed.

2. Common Block Variables

   /XCS/

   MEMCUR - name of the Mxxx member on XSUNIT in current execution.

   MXMDB - IDX of the Member Description Block Table (MDBT) residing in GDS.

   MNAME, MCUR, MCALL, MRL, MLL - position parameters for a Member Description Block (MDB) which is an entry in the MDBT.

   /XSPT/

   JCØN - the logical system parameter indicating the action to be taken by XMERR. If JCØN = .TRUE., then no action is taken. If JCØN = .FALSE., then the CS sequence is searched for a PRØCEED CS or the ENDCS whichever occurs first.

3. Control Structures

   MDBT - the Member Description Block Table (MDBT) residing in GDS. There is an MDB, or entry, for each Mxxx name assigned. The MDB contains descriptive and status information about the Mxxx.

Output:

1. Data Base Structures

   XSUNIT - all Mxxx members remain unchanged and are closed.

2. Common Block Variables

   MEMCUR - the name of the Mxxx in current execution resulting from action taken by XMERR. If a search was performed then Mxxx contains the first encountered PRØCEED or the ENDCS. If a search was not performed then MEMCUR is unchanged from the input value.

3. Control Structures

MDBT - if a search was not performed then all MDB entries are unchanged. If a search was performed, then the value of the contents of the MCUR position in the MDB of each Mxxx member involved in the search process has changed. This value in the MDB corresponding to MEMCUR is the position of the CS record which immediately preceeds the found PRØCEED or ENDCS CS record. (This is necessary for the CS Processing Phase to resume with the PRØCEED or ENDCS.) This value in the MDB corresponding to any other Mxxx involved in the search is the position of the last CS record in Mxxx which is now in completed execution.

Functional Description: The Error Processing Phase determines the action to be taken whenever a non-fatal error occurs during the processing of a CS. The error could have been encountered during the CS Processing Phase or during the F.M. Processing Phase. If it occurred in the former phase, then the CS was directly responsible for the error. If it occurred in the latter phase, which is always the result of an EXECUTE CS being processed, then the error was not directly caused by the EXECUTE CS but instead by the F.M. which was executed. In both cases, however, the CS last processed by the CS Processing Phase is considered by XMERR to be in error and the CS image is printed with an error message.

Further action to be taken is determined by the system parameter JCØN. If JCØN is set to .TRUE. on entry, then no further action is taken. Subsequently, when the CS Processing Phase is resumed by the driver XM, processing will continue with the CS immediately following the CS in error. If JCØN is set to .FALSE. on entry, then a search is performed to locate either the first PRØCEED CS after the CS in error or if no PRØCEED is found, then the ENDCS CS. Subsequently, when the CS Processing Phase is resumed by the driver XM, processing will resume with the PRØCEED or ENDCS.

The search for the PRØCEED begins with the CS following the CS in error. The search continues sequentially forward through the current CS set in execution. If during the search a CALL CS is encountered, it is not processed and the Secondary CS Stream specified is not brought into the search process. The CALL CS is skipped as any other CS. If the current CS Set is a Secondary CS Set and it is exhausted without a PRØCEED CS, then upon encountering the RETURN CS, the Mxxx containing the Secondary CS Set is closed and processing returns to the CS immediately following the CALL CS in the calling member. This procedure continues until a PRØCEED CS is detected or until processing returns to the Primary CS Set and an ENDCS CS is detected. On exit from XMERR, all Mxxx members used in the search are closed.

Logical Description: Immediately upon entry XMERR calls the XCSPM module to open the currently executing Mxxx member, allocate local core to receive a control statement record from the Mxxx member, and position the Mxxx member to the next CS record.

Upon return from XCSPM, the Mxxx member is positioned to the CS in error via the MMSKIP service module. The CS in error is gotten from the current Mxxx via the MMGETR service module. The CS record is echoed with an appropriate message telling the user that the current CS resulted in a system error and execution will continue with the next CS record or with the next PRØCEED CS, depending on the system parameter JCØN.

If the system parameter JCØN is set to .TRUE., indicating that execution should continue with the CS record following the CS in error, then the system parameter MEMCUR is unchanged and the number of the CS record in current execution on MEMCUR (the CS in error) is also unchanged. This insures that the CS record to be executed next by the Control Statement Processing Phase (XCSP) will be the CS immediately following the CS in error.

If the system parameter JCØN is set to .FALSE., this indicates that execution should not continue with the CS following the CS in error, but instead should continue with the next PRØCEED CS. If a PRØCEED CS is not detected, then execution should continue with the ENDCS CS.

In the case that JCØN is .FALSE., XMERR sequentially reads CS records from MEMCUR, bringing each into Local Dynamic Storage via MMGETR, until a PRØCEED or ENDCS CS is detected. In the event that a RETURN CS is detected, the current Mxxx is closed, system parameter MEMCUR is defined as the calling Mxxx, the calling Mxxx is opened, and the search continues in the calling member. Detection of a CALL CS in the current CS Set does not alter the course of the search. The CALL CS is skipped and the search continues with the CS immediately following the CALL. This procedure is iterated, and the number of the CS record in current execution on MEMCUR is incremented, until a PRØCEED or ENDCS CS is detected.

On exit from XMERR, the name of the current Mxxx member is defined by the system parameter MEMCUR, and the number of the CS record in current execution for the MEMCUR Mxxx has been defined dependent on the value of the system parameter JCØN.

<u>Error Philosophy</u>: The XMERR module aborts only if an error is detected by MMGETR while trying to get the next CS record from the current Mxxx member.

3.5.4.6 Secondary Edit Phase (XCA)

Purpose: The Secondary Edit Phase module (XCA) is called by the Control Statement Processing Phase (XCSP) to process a CALL control statement. If this is the first execution of the CALL CS, XCA builds an Mxxx member containing CS records that correspond to the control statements in the Secondary Input Stream. If the Mxxx member is built successfully, or if the Mxxx has been previously executed, the Secondary Edit Phase provides the environment required for the CS Processing Phase to resume execution with the first control statement on the new Mxxx member.

Input: Primary Input to the Secondary Edit Phase is the Secondary Input Stream residing in card image (CI) format on the member specified as DU(DM) on the CALL CS. Pertinent input is described below. Other input required for processing but not necessary for understanding, is not included.

1. Data Base Structures

   DU(DM) - the data member specified by the DU(DM) on the CALL CS contains the Secondary Input Stream in CI format.

2. Common Block Variables

   /XCS/

   MEMCUR - the name of the Mxxx member in current execution on entry to XCA is defined by MEMCUR. MEMCUR names the Mxxx member where the CALL CS resides.

   /XCVT/

   NERR - the executive system logical error indicator NERR is always .FALSE. on entry to XCA, indicating that no errors have been detected in processing.

3. Control Structures

   MDBT - the Member Description Block Table resides in GDS and is a system table type 1. The MDB entry for the Mxxx member being called into execution exists in the MDBT in initialized or executable format. The values in an initialized MDB indicate that the Mxxx does not exist, whereas the values in an executable MDB indicate that the Mxxx has been constructed.

Output:

1. Data Base Structures

   Mxxx - the first time a CALL CS is executed, the Secondary Edit Phase validates and builds the Mxxx member being called into execution. Mxxx contains a variable length control statement record for each complete CS edited in the Secondary Input Stream and a Label Record that provides a cross-reference to each labeled CS on Mxxx. The member is in a format recognized by the CS Processing Phase. If an error is detected in the Secondary Input Stream, writing to the Mxxx member is suppressed but editing continues until the Secondary Input Stream has been exhausted.

2. Common Block Variables

/XCS/

MEMCUR - the name of the current Mxxx type member in execution. If the CALL CS has been processed without error, MEMCUR indicates the Mxxx to which control has been transferred and the Mxxx in execution on entry has been closed. But if errors have been detected in processing the CALL CS, MEMCUR is unchanged from its entry value.

/XCVT/

NERR - the executive system logical error indicator. If an error is detected in processing the CALL CS, NERR is set to .TRUE. on exit.

3. Control Structures

MDBT - the first time a CALL CS is executed and an Mxxx member is built, a Member Description Block entry (MDB) is put into executable format for the Mxxx built.

Functional Description: The XCA module performs all operations necessary to process a Secondary Input Stream as a result of a CALL CS. If it is the first execution of the CALL CS, then an Mxxx type member must be built from the card images in the Secondary Input Stream. The Secondary Input Stream resides on the member specified by the DU(DM) field on the CALL CS image. The name of the Mxxx member to be built is found in the CALL CS record.

The Mxxx member is built in a single sequential pass on the Secondary Input Stream. As each CS in the Secondary Input Stream is processed, substitutions are made in the CS image from optional replacement names specified on the CALL CS.

Once substitutions have been made, the new CS image is used to build an unformatted, variable length CS record. CS records for the Secondary CS Set are edited and built in the same manner used to build CS records for the Primary CS Set.

Certain CS names, or forms of a CS, that were valid in the Primary Input Stream are not valid in the Secondary Input Stream. The SØURCE=* form of the UPDATE and TABLE control statements are not valid to the Secondary CS Set. Neither is the DATA CS valid to the Secondary CS Set.

A CALL CS is processed the same in the Primary Input Stream and the Secondary Input Stream. An Mxxx member name is assigned and an MDB entry initialized each time a CALL CS is encountered.

If an end-of-member condition is detected, a RETURN CS is simulated for internal control. Detection of the RETURN CS indicates the end of the Secondary Input Stream.

As the Mxxx member is built, the MDB for the Mxxx is put into executable format.

If the Mxxx member is built successfully, or if the Mxxx was built and executed previously (due to a previous processing of the CALL CS), then XCA transfers control to the new Mxxx so that the CS Processing Phase (XCSP) will resume execution with the first CS record on the new Mxxx. In order to provide such an environment, the following steps must be done:

1. Close the Mxxx member that was in execution on entry to XCA.
2. Open the new Mxxx to read and position to the first CS record on the member.
3. Free LDS blocks for the Mxxx member that has been closed and re-allocate LDS blocks as required for processing the new Mxxx member.
4. The MEMCUR parameter that defines Mxxx member in current execution is set to name of new Mxxx.

If the Mxxx member was not built successfully, then the entry environment is unchanged. Control is not transferred to the new Mxxx. Before exit, an error indicator is set to inform the caller that an error was detected in the Secondary Edit Phase.

Logical Description: Immediately upon entry to the XCA module, the XCAI module is called to open the member containing the Secondary Input Stream and the Mxxx member to be built. In addition, XCAI allocates expandable LDS blocks necessary for building the Mxxx Label Reference Table (LRT) and Label Record Table (LREC), and fixed length LDS blocks for building the Substitution Table (LSUB) and the new CS Image Block (NCSIB).

The XCA module then calls XCABST to build the Substitution Table from the replacement values, if present, specified on the CALL CS. If at least one replacement set is specified on the CALL CS, XCABST cracks the CALL CS image without converting fields (XCRWC). Replacement sets appear in the form oldvalue = newvalue on the CALL CS. To build the Substitution Table, the = fields are stripped out and only the old- and newvalue fields are retained. Upon completion, the Substitution Table contains the type code and corresponding oldvalue field and the type code and corresponding newvalue field for each replacement set.

Once the Substitution Table is complete, the following process is iterated until the Secondary Input Stream has been exhausted (a RETURN CS detected or simulated):

> The XCANCS module is called to produce a new CS image by getting the next CS image from the Secondary Input Stream and making field substitutions as required by the CALL CS. Field substitutions are made according to values in the Substitution Table previously built on entry to XCA. Any field type may be replaced by any other field type. The comment portion of the original CS, if present, is retained. If the new CS image with substitution causes the comment portion to overflow a card image, then the comment portion is truncated accordingly. If the CS image without comment exceeds the maximum allowable card images, then an end-of-data character is simulated and the CS is truncated.
>
> If an end-of-member condition is detected before a RETURN CS is detected, a RETURN CS is simulated for internal control. If an end-of-member condition is detected on an incomplete CS, then that CS is replaced by a RETURN CS.
>
> If the current CS is found to be in error the system parameter JECHØ is automatically turned on so that subsequent images will be echoed. A CS is considered in error if the original CS image exceeds maximum allowable images or if the new CS image without comment portion exceeds maximum allowable images.
>
> When the new CS image is complete, the XCAMXX module is called to build and validate a control statement record valid for a Secondary CS Set. The CS record in built in the same manner used to build control statement records for the M001 root member. XRT sub-modules are called to perform the following functions:
>
> 1. Update label record table for Mxxx being built (XRTBLR)
> 2. Update label reference table for Mxxx being built (XRTLRF)

3. Perform syntax check according to format requirements for particular CS (XRTSYN)
4. Simulate CS complete condition if CS image exceeds maximum cards per CS with no valid CS terminator (XRTTC)
5. Get CS record into format ready to be written on new Mxxx (XRTBCS)

Comment cards (CS where first non-blank character is end-of-data character) are included on the new Mxxx as a CS with CØNTINUE substituted as the CS name.

The XCAMXX module allocates and initializes an MDB entry in the MDBT for each CALL CS processed. The XRT sub-module XRTCAL, is called to form the new Mxxx name assigned to the CALL CS and initialize the MDB.

If a CS error is detected, the Mxxx member is considered to be in error. NERR is set to .TRUE. and writing to the Mxxx member is inhibited, although editing and building CS records continues. A CS is considered in error under any of the following circumstances:

1. Unrecognizable field detected on CS image
2. Invalid label field (either invalid form or duplicate labels)
3. Invalid CS name
4. Invalid syntax check for CS name
5. Maximum images (MAXCC) exceeded

Once the Mxxx member has been built, the XRT sub-module XRTLSA is called to insure that all label references on the member are satisfied. If all labels have been satisfied and the Mxxx member is error free, then the label record is written on the Mxxx member as the last record.

The XCACLØ module is then called to perform the closing functions. XCACLØ frees the Substitution Table (LSUB), the New CS Image Block (NCSIB), the Label Record Block (LREC), and the Label Reference Table (LRT) in Local Dynamic Storage. XCACLØ also closes the Secondary Input Stream member and the new Mxxx member just built.

XCA then completes the MDB entry in the MDBT for the new Mxxx member by inserting in the MDB the name of the Mxxx that called the new Mxxx member just built. The calling member is the Mxxx member that was in current execution on entry to XCA.

If the new Mxxx member was successfully built, or if the Mxxx member was previously executed then the XCATRA module is called to transfer execution control to the new Mxxx.

XCATRA first closes the Mxxx member that was in current execution on entry to XCA. Then Local Dynamic Storage blocks are freed and re-allocated according to the requirements for the new Mxxx. The new Mxxx is opened to read and the label record is validated and moved to the newly allocated LDS Label Record Table. Then the new Mxxx is positioned to read the first CS record.

If no errors have been detected on exit from XCA, then XCATRA has set up the appropriate environment such that XCSP will resume execution with the first CS record on the new Mxxx member.

Error Philosophy: The Secondary Edit Phase aborts via the XXFMSG fatal message writer if an error is detected when opening the new Mxxx member to be written.

The Substitution Table is allocated for exact requirements of replacement values specified on the CALL CS. An error is indicated (possibly in the replacement fields specified on the CALL CS) if the number of words moved to the table in building does not match allocated table length. In such a case, XCABST aborts via XXFMSG fatal message writer.

When building a CS record for the new Mxxx member, XCAMXX aborts via XXFMSG if the CS record block overflows because the allocated length was not the maximum required or if an unexpected Member Manager return status is detected while reading the Secondary Input Stream Member.

Several other errors are not immediately fatal but do result in ultimate termination of processing at the end of the Secondary Edit Phase. If the Secondary Input Stream member does not exist or is not in card image (CI) format, logical error indicator NERR is set and a message is printed. Also, the error indicator is set and message printed if LDS is insufficient to allocate all of the tables required for processing.

Edit errors detected in the Secondary Edit Phase cause writing on the new Mxxx member to be suspended. However, editing and building CS records continues until the Secondary Input Stream has been completely processed. Error messages are printed before the corresponding CS record is echoed. If a CS error is detected, the system parameters NERR and JECHØ are set to .TRUE. and the current CS is echoed.

A CS image that exceeds the maximum card images (MAXCC) per CS falls under the category of edit errors above. The control statement is arbitrarily terminated at the end of the last allowable image and processing continues as for a valid CS. An incomplete CS detected at an end-of-member is not processed, but instead is replaced by a RETURN CS which is processed as a valid CS.

3.5.4.7 Normal Termination Phase (XEN)

Purpose: The EM module XEN is called during the Control Statement (CS) Processing Phase by XCSP to process the control statement ENDCS. The ENDCS indicates that the set of control statements provided by the user as card image input to ANOPP (i.e., the Primary Input Stream) has been completely processed and ANOPP termination is desired. The process of terminating ANOPP upon normal completion of processing is called the Normal Termination Phase and is controlled by XEN.

Input: The M001 member on unit XSUNIT, which contains the executable form of the Primary Input Stream, is open to read.

Output: There is no output since ANOPP is terminated.

Functional Description: The Normal Termination Phase includes printing an informative message indicating ANOPP normal termination, closing the opened member M001, and halting further execution.

Logical Description: XEN is a simple module requiring no calls to lower level modules. The required message is printed, M001 is closed via a MM call, and execution is halted via the FØRTRAN STØP command.

Error Philosophy: No error condition is encountered.

3.5.4.8 Error Termination Phase (XXFMSG)

Purpose: The Error Termination Phase is controlled by the Executive Management System (EM) auxiliary module XXFMSG. It is called by any EM module which detects an error condition which inhibits further meaningful execution (i.e., a fatal error).

Termination of ANOPP will result with an informative message as to the cause.

Input: The calling module via an argument list provides the calling module name, defines the message number to be printed, and provides additional descriptive information. The same message number may be requested by several calling modules with varying descriptive information. Four arguments are provided for descriptive information with usage dependent upon the message. The message number range is 1-999. See Appendix C.

Output: Message text on ANOPP output file including specified error message and a traceback via XEXIT.

Functional Description: XXFMSG identifies and prints the message requested. Messages have two parts. The first part of all messages is fixed as follows:

```
*** EXEC ERRØR (ERRØR NUMBER --- )   *** ( CALLER --- )
```

The second part of all messages is unique for each message number and describes the cause of error.

A traceback which prints module names from the calling module to the driver XM is provided and ANOPP is then terminated.

Logical Description: Identification and message printing is performed directly by XXFMSG. FØRTRAN WRITE statements with pre-defined FØRMATS are utilized. The General Utilities XTRACE and XEXIT are called to perform the traceback and the ANOPP termination respectively.

Error Philosophy: The message number is validated upon entry with no further possibility of error occurrence.

3.5.5 Auxiliary Modules

An auxiliary modules does not perform a function which is unique to a specific executive phase or group of EM modules but instead performs a function common to many EM modules during various executive phases. It is a general purpose module available for usage only by other EM modules.

3.5.5.1 Fatal Error Message Writer (XXFMSG)

Purpose: The XXFMSG module is utilized to print an informative error message whenever a fatal error is encountered and detected by an EM module and to terminate ANOPP.

XXFMSG controls the Error Termination Phase of EM and is discussed in Section 3.5.4.8. It is called by many EM modules during the various executive phases whenever an error condition which inhibits further meaningful execution is detected.

For full description of this module, see Section 3.5.4.8.

3.5.5.2 Non-Fatal Error Message Writer (XXNMSG)

Purpose: The XXNMSG module is utilized to print an informative error message whenever a non-fatal error is encountered and detected by an EM module.

Input: The calling module via an argument list provides the calling module name, defines the message number to be printed, and provides additional descriptive information. The same message number may be requested by several calling modules with varying descriptive information. Four arguments are provided for descriptive information with usage dependent upon the particular message. The message number range is 1001-1999.

Output: There is no output upon return to the calling module.

Functional Description: XXNMSG identifies and prints the message required. Messages have two parts. The first part of all messages is fixed as follows:

```
*** EXEC ERRØR ( ERRØR NUMBER --- )   ***  ( CALLER --- )
```

The second part of all messages is unique for each message number and describes the cause of error.

Logical Description: Identification and message printing is performed directly by XXNMSG. FØRTRAN WRITE statements with pre-defined FØRMATS are utilized.

Error Philosophy: The message number is validated upon entry and the Error Termination Phase is invoked via the XXFMSG auxiliary if found to be invalid.

3.5.6 Hierarchy Charts

A hierarchy chart is a graphical representation of the logical relationship between modules. Figures 1-24 are the hierarchy charts for the Executive Management System (EM).

In general, only EM modules appear as a block entity in the charts and all EM modules appear at least once. The charts are in alphabetical order with respect to module name except for Figure 1 which is the hierarchy chart for the driver XM from which all other EM modules, except EM auxiliary modules, derive. A hierarchy chart for each auxiliary module is also among the alphabetized charts.

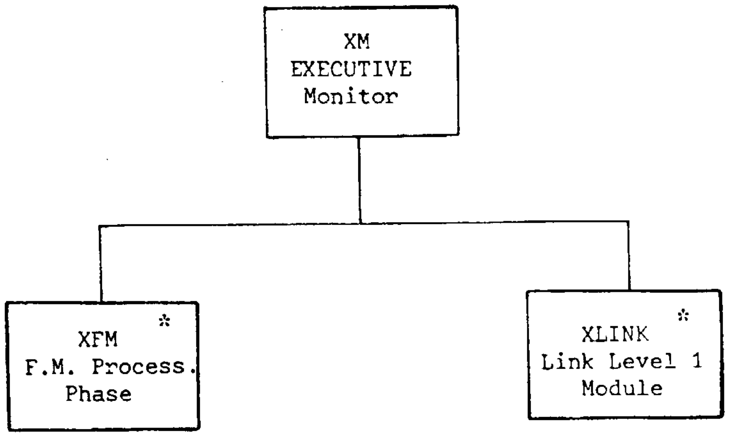
A module which is not part of EM but is called by an EM module is, in general, not shown as a block entity but is listed at the bottom of the chart. The module may be an ANOPP executive module which is part of the Data Base Management System (DBM), the Dynamic Storage Management System (DSM), or the General Utilities. It also may be a subprogram provided by one of the CDC operating system libraries. In either case, the module is generally of a service or utility nature and may be called many times by various EM modules. One of these service type modules may, however, be of sufficient design importance to the calling EM module that it should receive more emphasis than simply being listed. In these cases, the non-EM module is represented as a block entity for logical emphasis and is noted as such on the chart.

Symbols and heads used in the hierarchy charts are given below:

NAME purpose	NAME - module name purpose - brief description
———————	indicates lower module is called by the higher module
*	in upper right corner of module block indicates module is expanded as a separate hierarchy

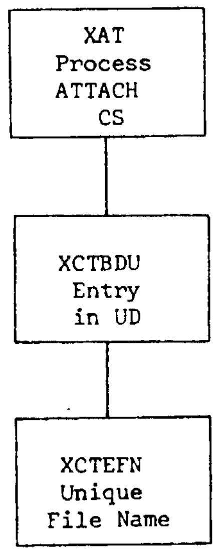
ANOPP Modules Called: a list of DBM, DSM, and General Utility Modules called by the modules in this figure

CDC System Library Subprograms Called: a list of subprograms called by the modules in this figure and which are not part of ANOPP but are provided by CDC NØS operating system libraries



ANOPP Modules Called: None

Figure 1. XM Hierarchy Chart

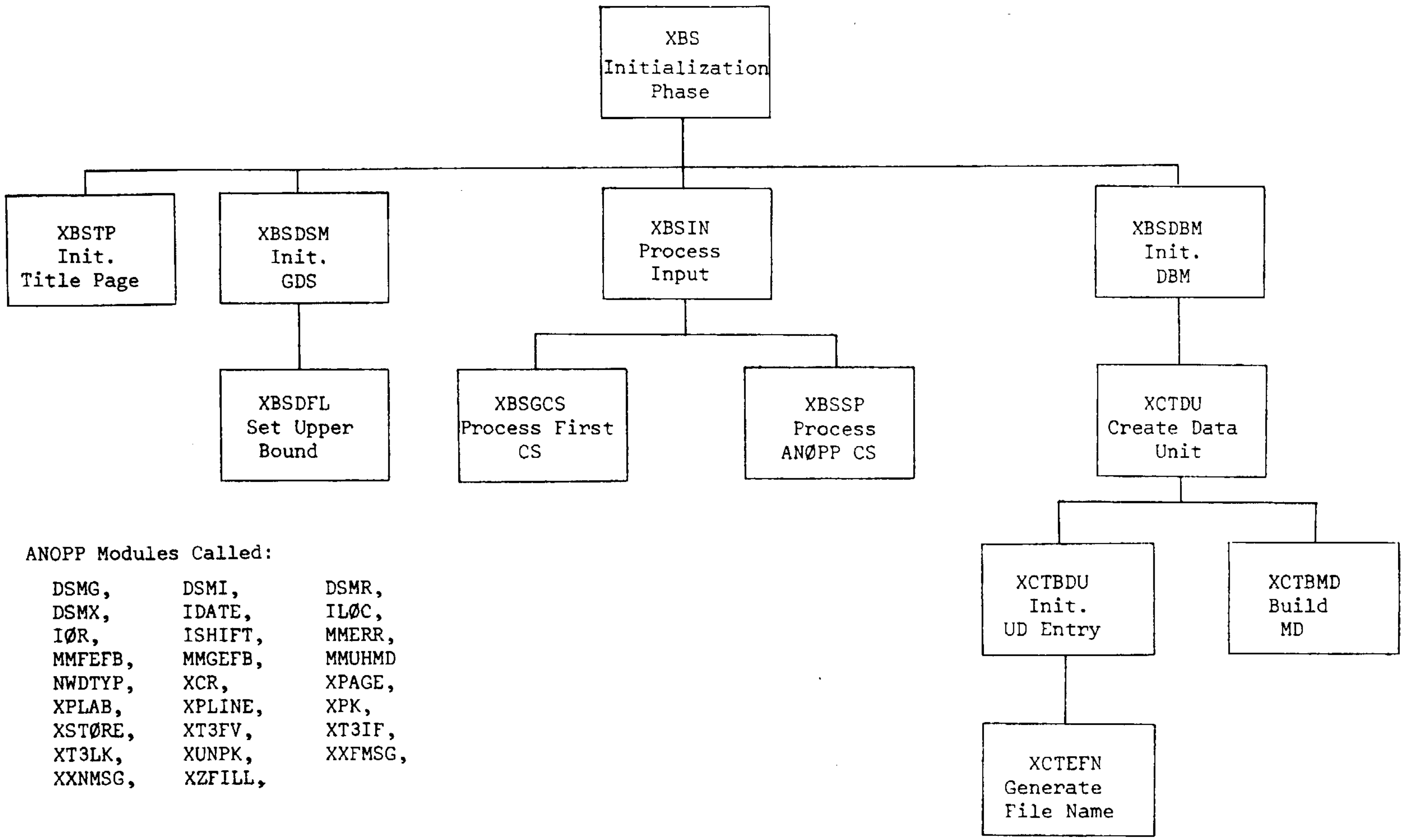


ANOPP Modules Called:

IØR, ISHIFT, MMCRMX, MMERR, MMFEFB,
MMGEFB, XT3FV, XT3LK, XZFILL

CDC System Library Subprograms Called: GET

Figure 2. XAT Hierarchy Chart



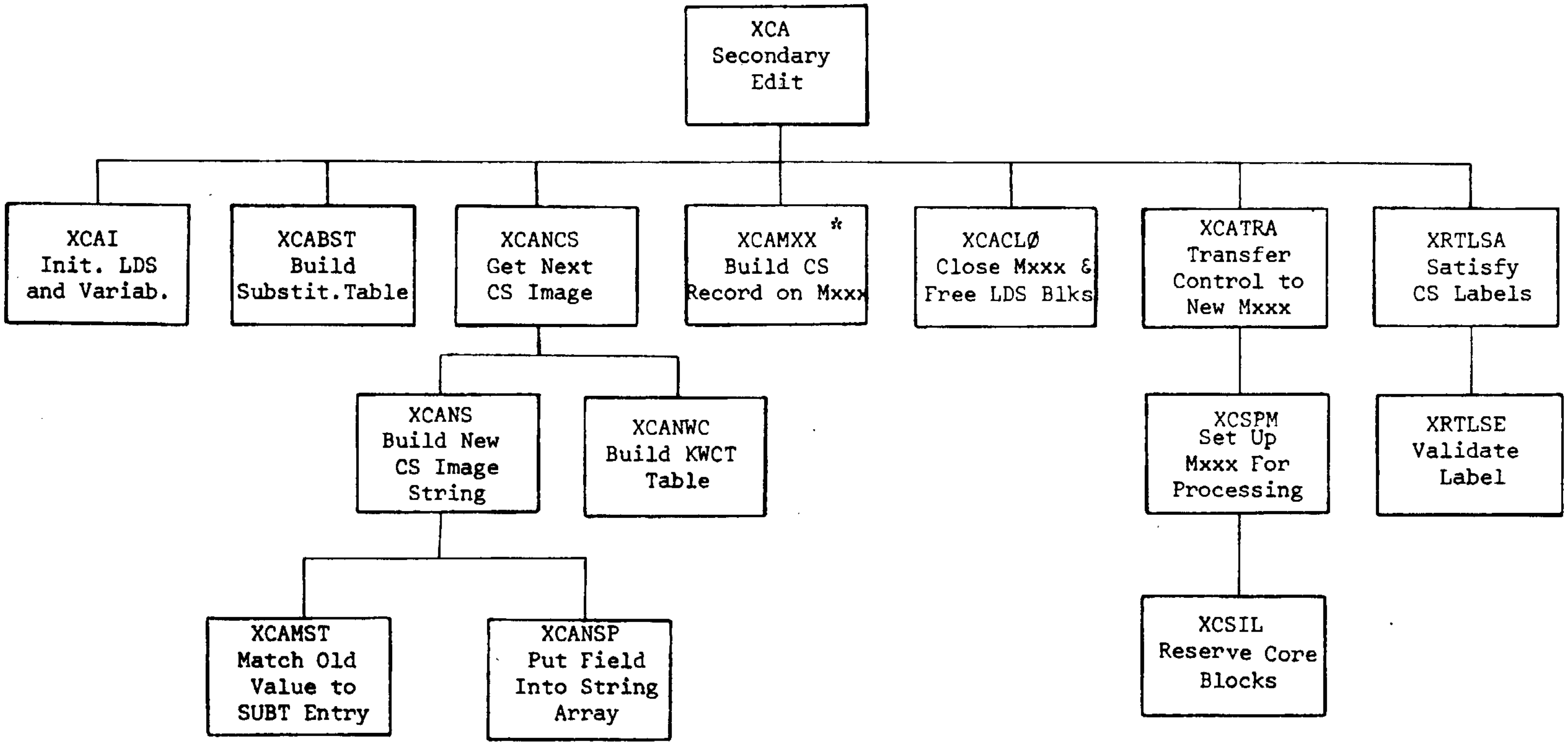
ANOPP Modules Called:

DSMG,
DSMX,
IØR,
MMFEFB,
NWDTYP,
XPLAB,
XSTØRE,
XT3LK,
XXNMSG,
DSMI,
IDATE,
ISHIFT,
MMGEFB,
XCR,
XPLINE,
XT3FV,
XUNPK,
XZFILL,
DSMR,
ILØC,
MMERR,
MMUHMD
XPAGE,
XPK,
XT3IF,
XXFMSG,

CDC System Library Subprograms Called:

EØF

Figure 3. XBS Hierarchy Chart

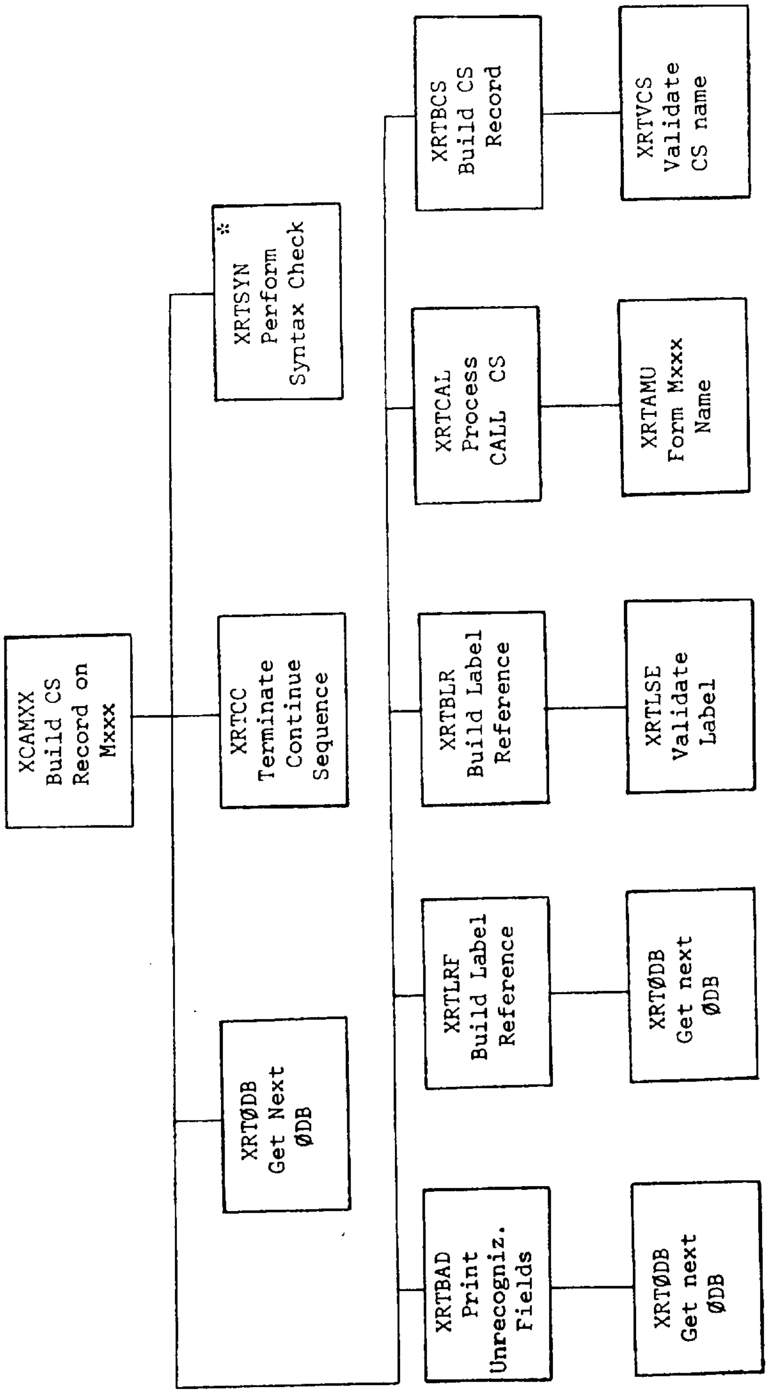


ANOPP Modules Called:

DSMF, DSMG, ICD, ICI, MMCLØS, MMGETR, MMØPRD, MMØPWD,
MMPØSN, MMPUTR, NWDTYP, MEMNUM, XCRWC, XMØVE, XFMTQ, XPKM,
XUNPKM, XUNPKT, XPLAB, XPLABQ, XPLINE, XXFMSQ, XXNMSG

Note$_1$. Module XCRWC is included as a block entity for emphasis but is expanded as a General Utility hierarchy in Section 3.9.4.

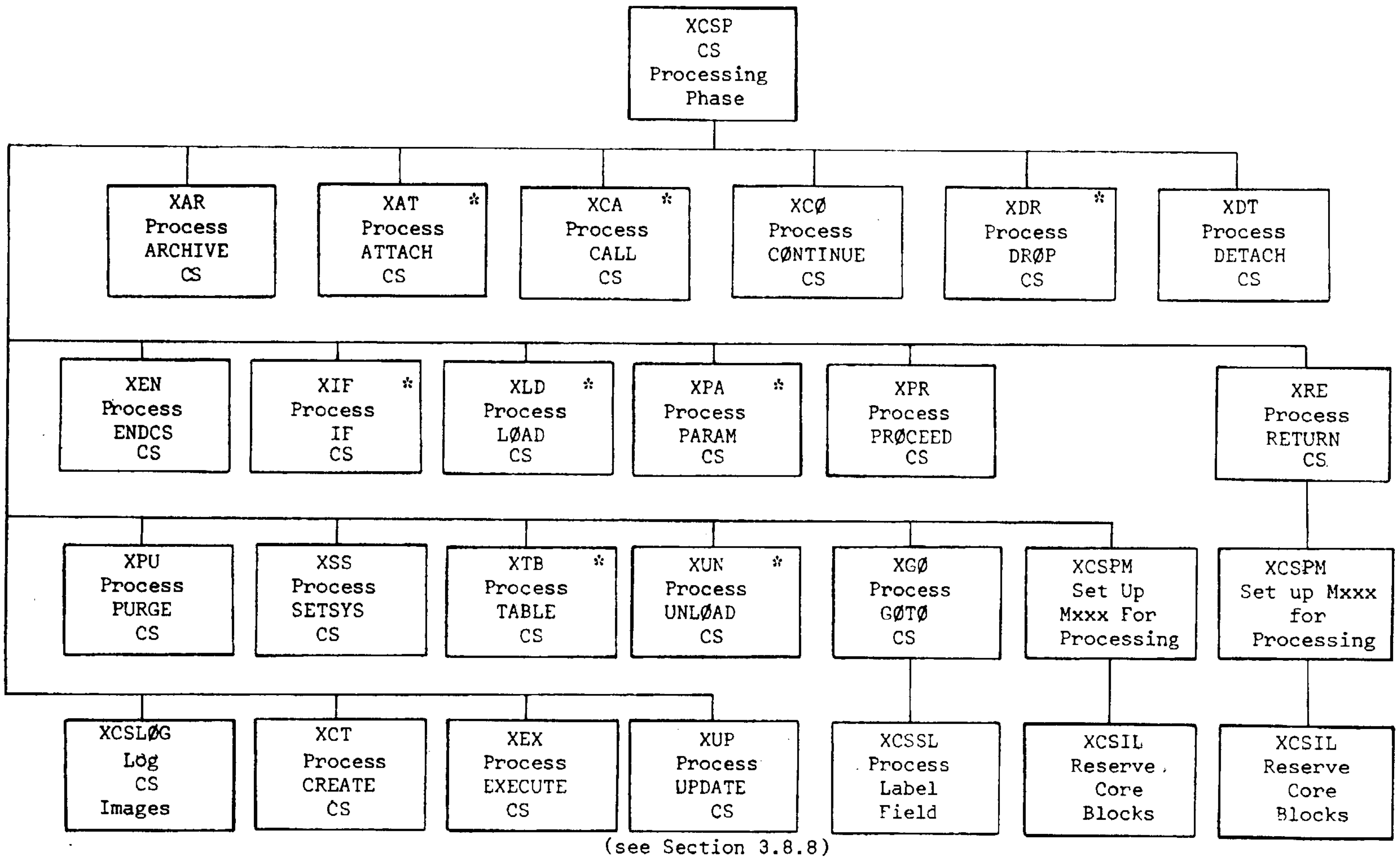
Figure 4. XCA Hierarchy Chart



ANØPP Modules Called:

DSMF, DSMG, DSMX, ICI, MMPUTR, NWDTYP,
XCR, XMØVE, XPK, XPLINE, XT1AL, XXFMSG, XXNMSG

Figure 5. XCAMXX Hierarchy Chart

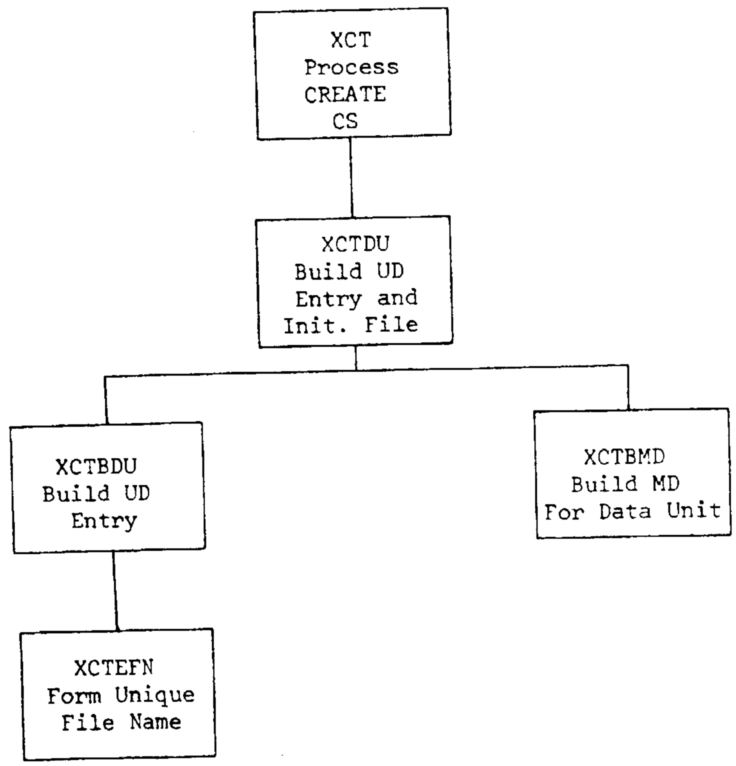


(see Section 3.8.8)

ANØPP Modules Called:

DSMB, DSMF, DSMG, DSMI, DSMR, MMCLOS,
MEMNUM, MMCRMX, MMERR, MMFEFB, MMGEFB, MMGETR,
MMOPRD, NWDTYP, XCSSL, XFETCH, XPLAB, XPLINE,
XT3FV, XT3LK, XXFMSG, XXNMSG

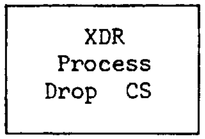
Figure 6. XCSP Hierarchy Chart



ANOPP Modules Called:

DSMX, IOR, ISHIFT, MMERR, MMFEFB,
MMGEFB, MMUHMD, XT3FV, XT3LK, XZFILL

Figure 7. XCT Hierarchy Chart



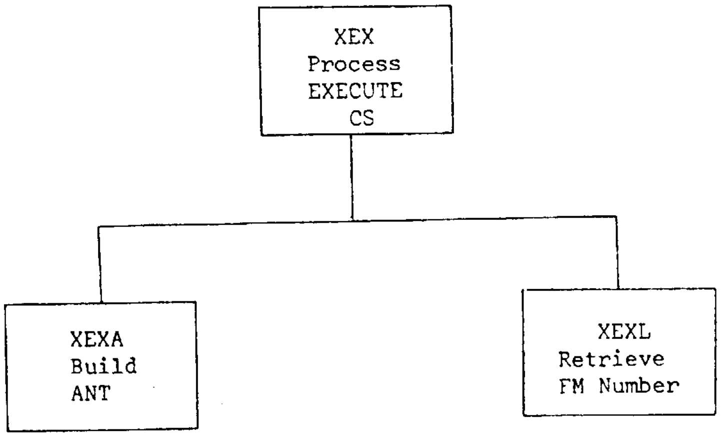
ANOPP Modules Called:

XPLINE, XPURGE, XT1FV

CDC System Library Subprogram Called:

FILESQ, ØPENM

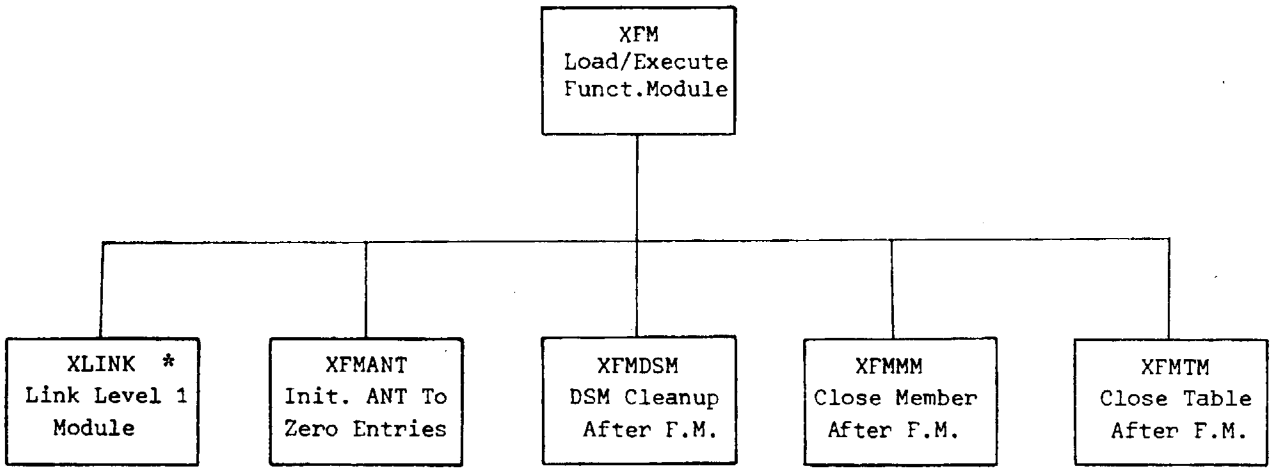
Figure 8. XDR Hierarchy Chart



ANOPP Modules Called:

DSMX, XXFMSG, XXNMSG

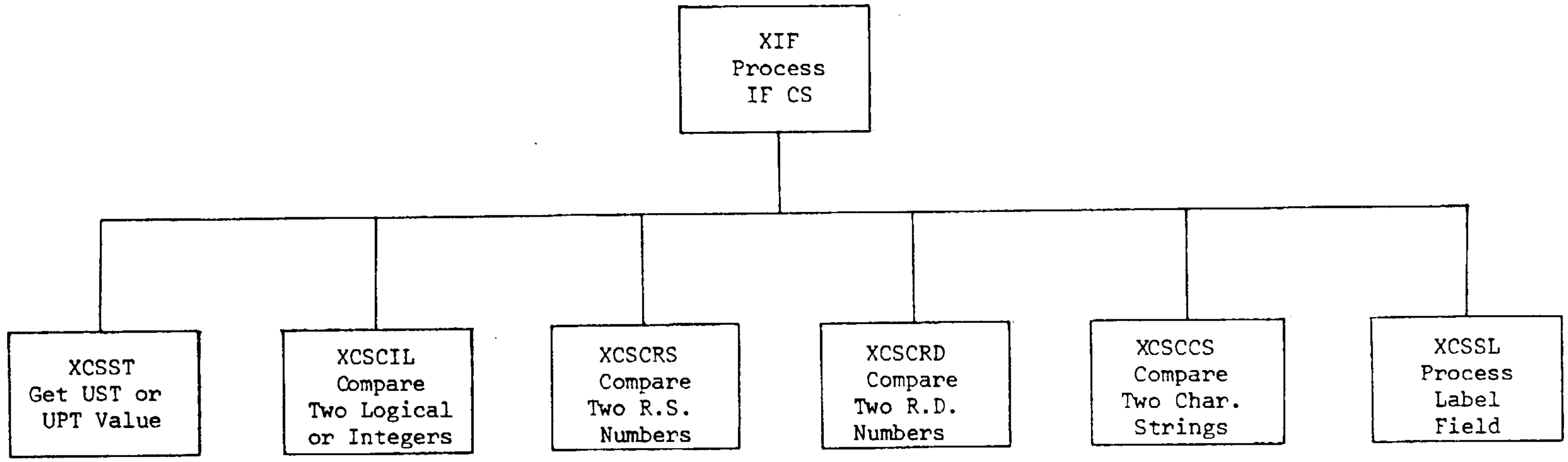
Figure 9. XEX Hierarchy Chart



ANOPP Modules Called:

```
DSMF,    DSMG,    DSMR,    DSMS,
MMCLSE,  MMDØMC,  TMFTE,   XPLAB,
XPLINE,  XXFMSG,  XXNMSG
```

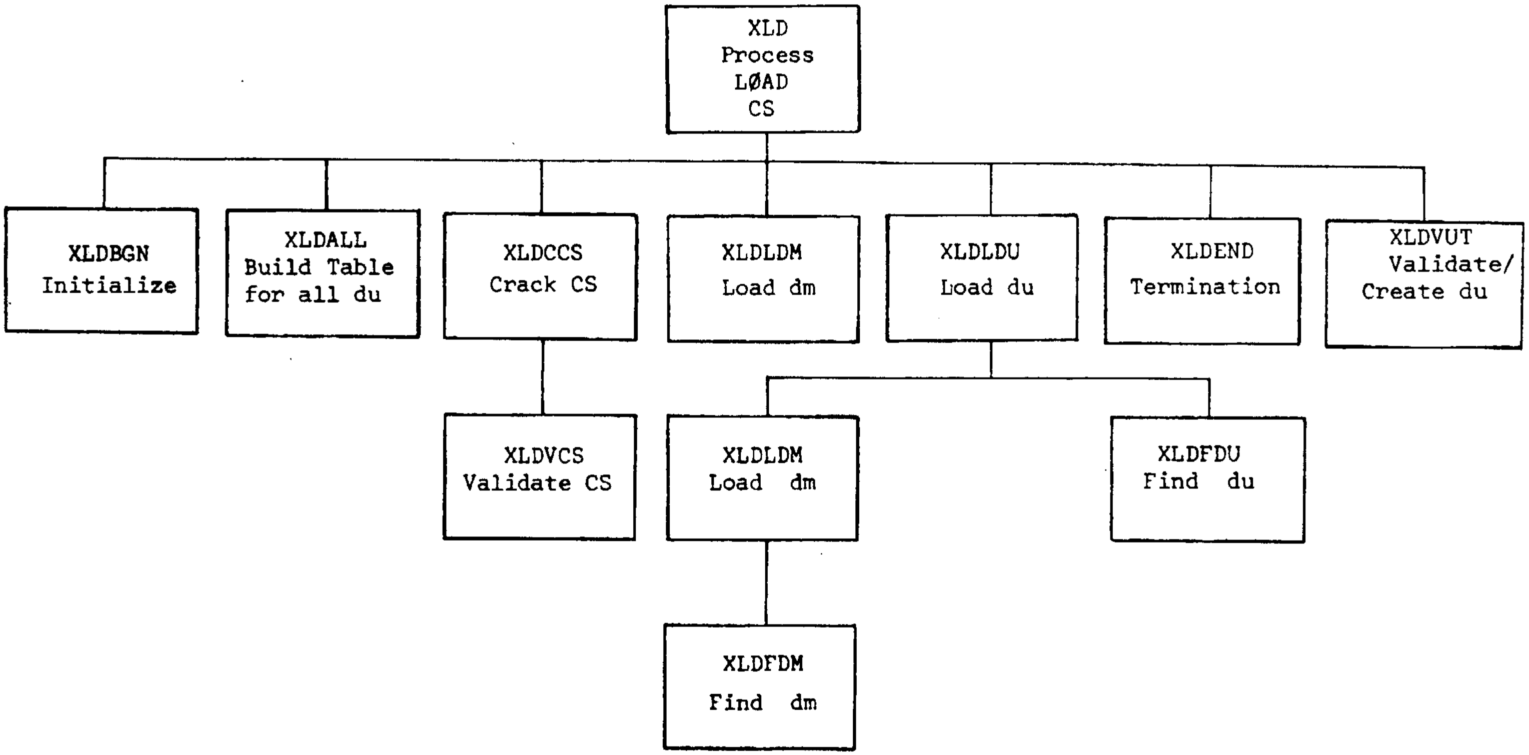
Figure 10. XFM Hierarchy Chart



ANOPP Modules Called:

MMPØSN, NWDTYP, XASKP,
XXFMSG, XXNMSG

Figure 11. XIF Hierarchy Chart



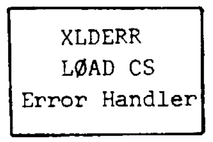
ANOPP Modules Called:

DSMF, DMSG, DSMQ, DSMX,
IAND, IFETCH, ILØC, MMDØMC,
MMIØMC, MMRMD, MMSUD, MMUHMD,
MMVBA, XLDERR, XSØRTF, XT1FV,
XT3FV XZFILL

CDC System Library Subprograms Called:

CLØSEM, FILESQ, GET, GETP,
ØPENM, PUT, SKIP

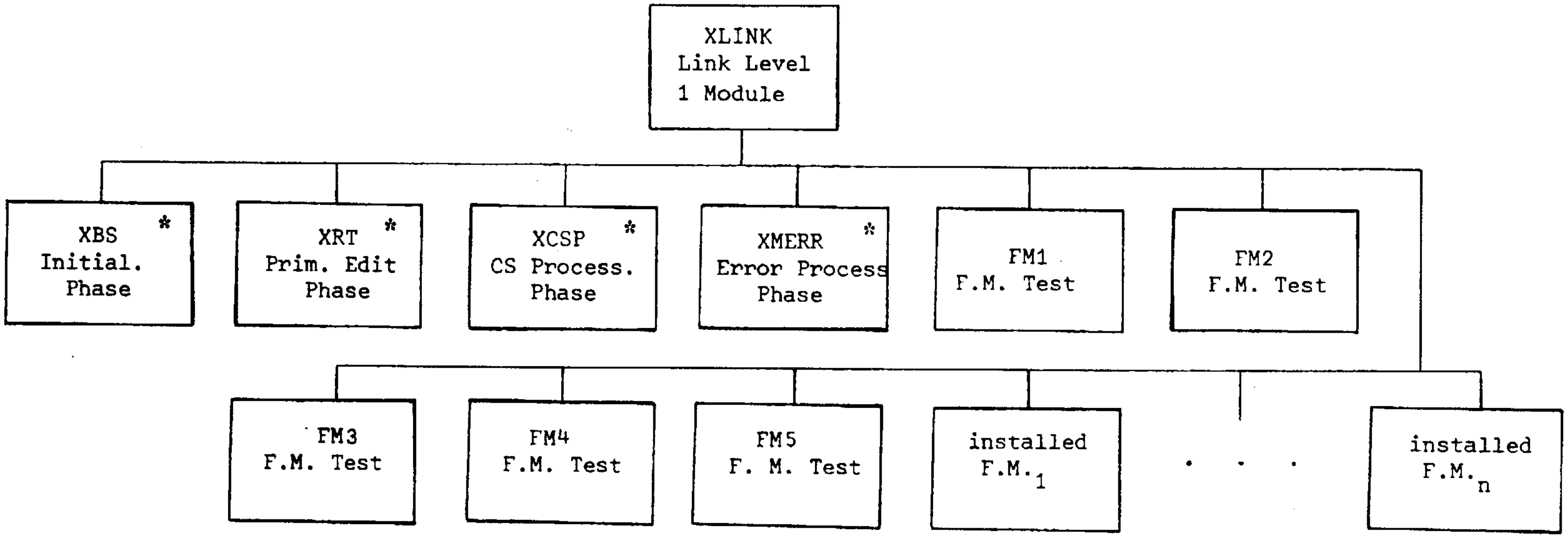
Figure 12. XLD Hierarchy Chart



ANOPP Modules Called:

XEXIT, XFETCH, XPLINE, XTRACE

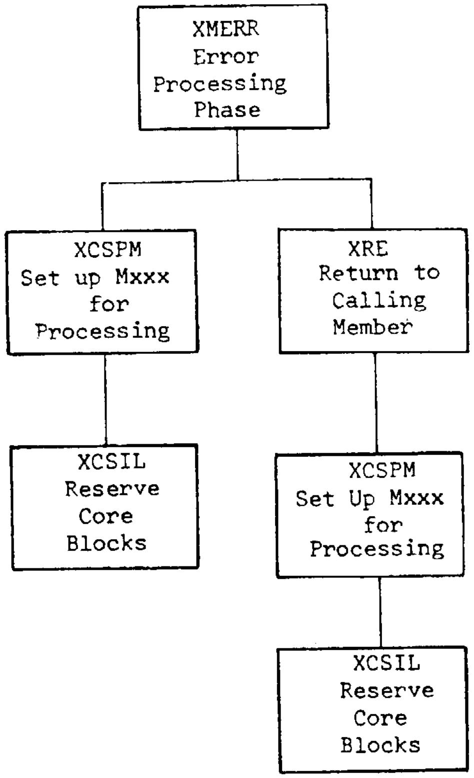
Figure 13. XLDERR Hierarchy Chart



ANOPP Modules Called:

XXFMSG

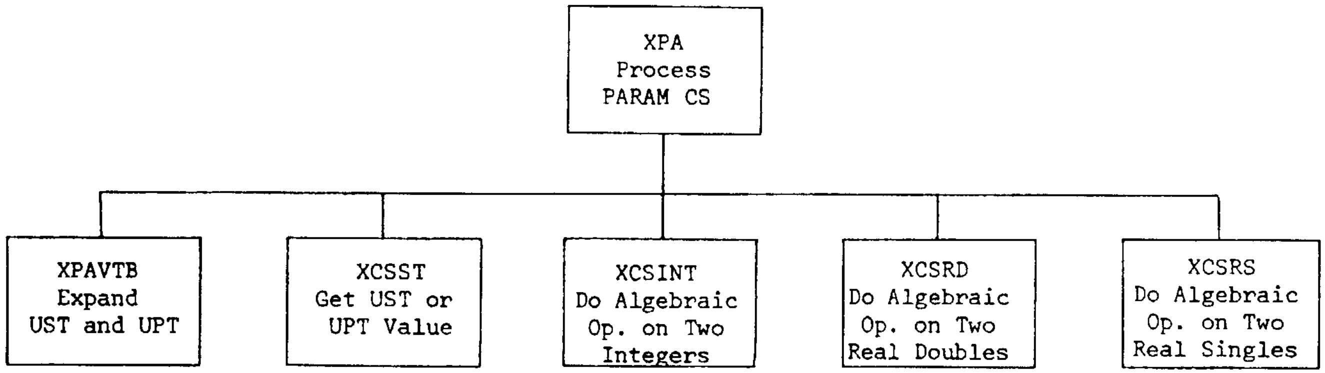
Figure 14. XLINK Hierarchy Chart



ANØPP Modules Called:

DSMB, DSMG, DSMI, DSMR, MEMNUM,
MMCLØS, MMGETR, MMØPRD, MMPØSN,MMSKIP,
XPLAB, XPLINE, XSTØRE, XXFMSG

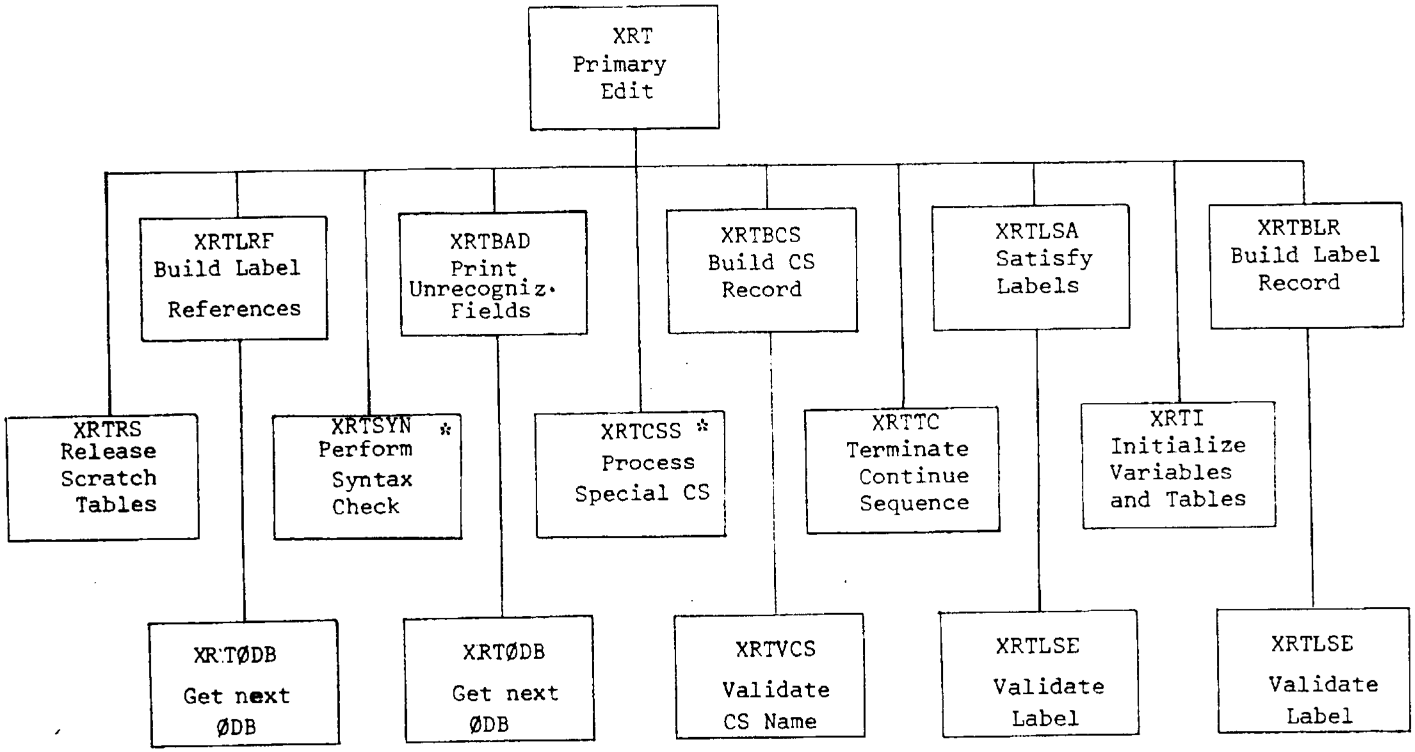
Figure 15. XMERR Hierarchy Chart



ANOPP Modules Called:

DSMX,	NWDTYP,	XASKP,
XPUTP,	XXFMSG,	XXNMSG

Figure 16. XPA Hierarchy Chart

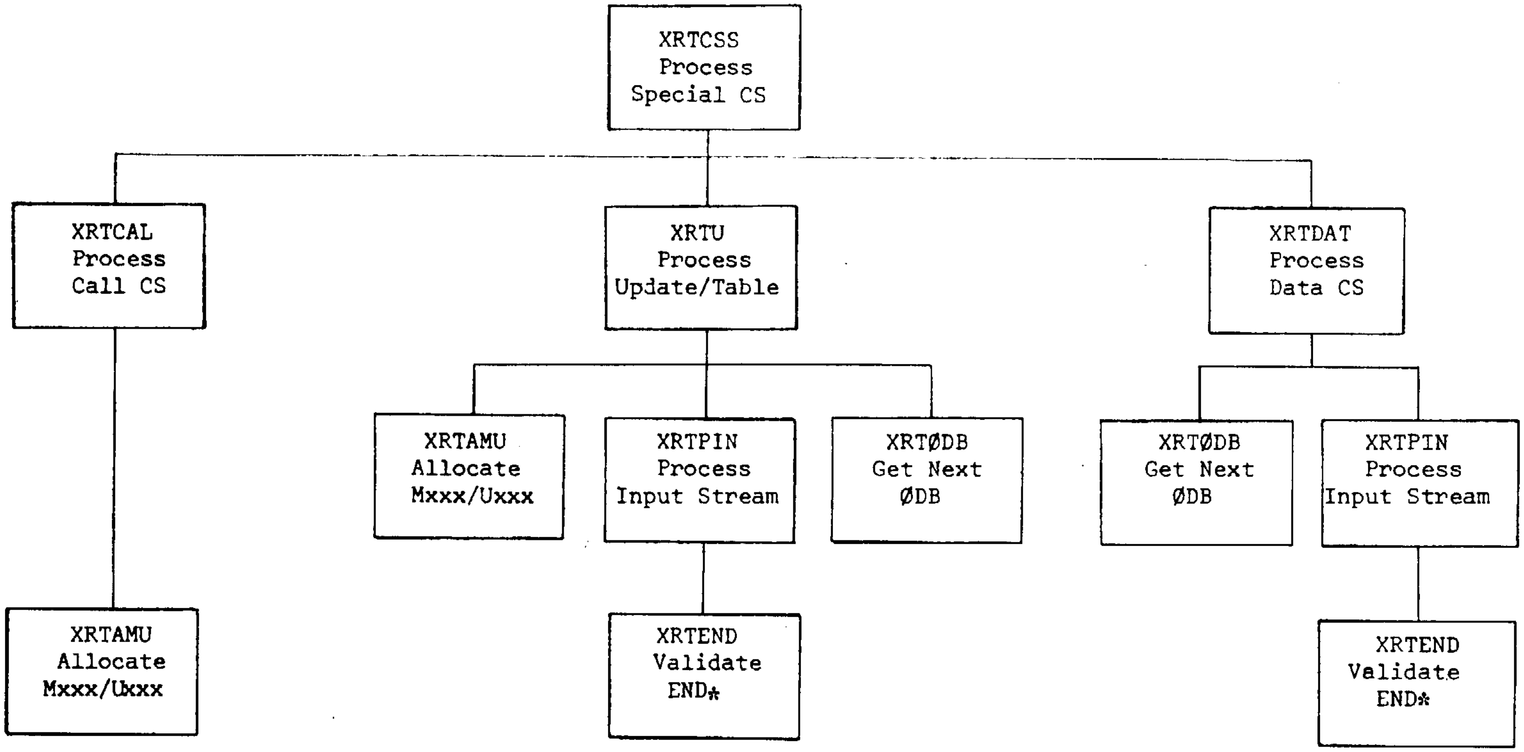


ANØPP Modules Called:

```
DSMB,     DSMF,     DSMG,     DSMI,     DSMR,     DSMX,
MMCLØS,   MMØPWS,   MMPUTR,   NWDTYP,   XCR,      XPAGE,
XPLAB,    XPLINE,   XXFMSG,   XXNMSG
```

CDC System Library Subprograms Called: EØF

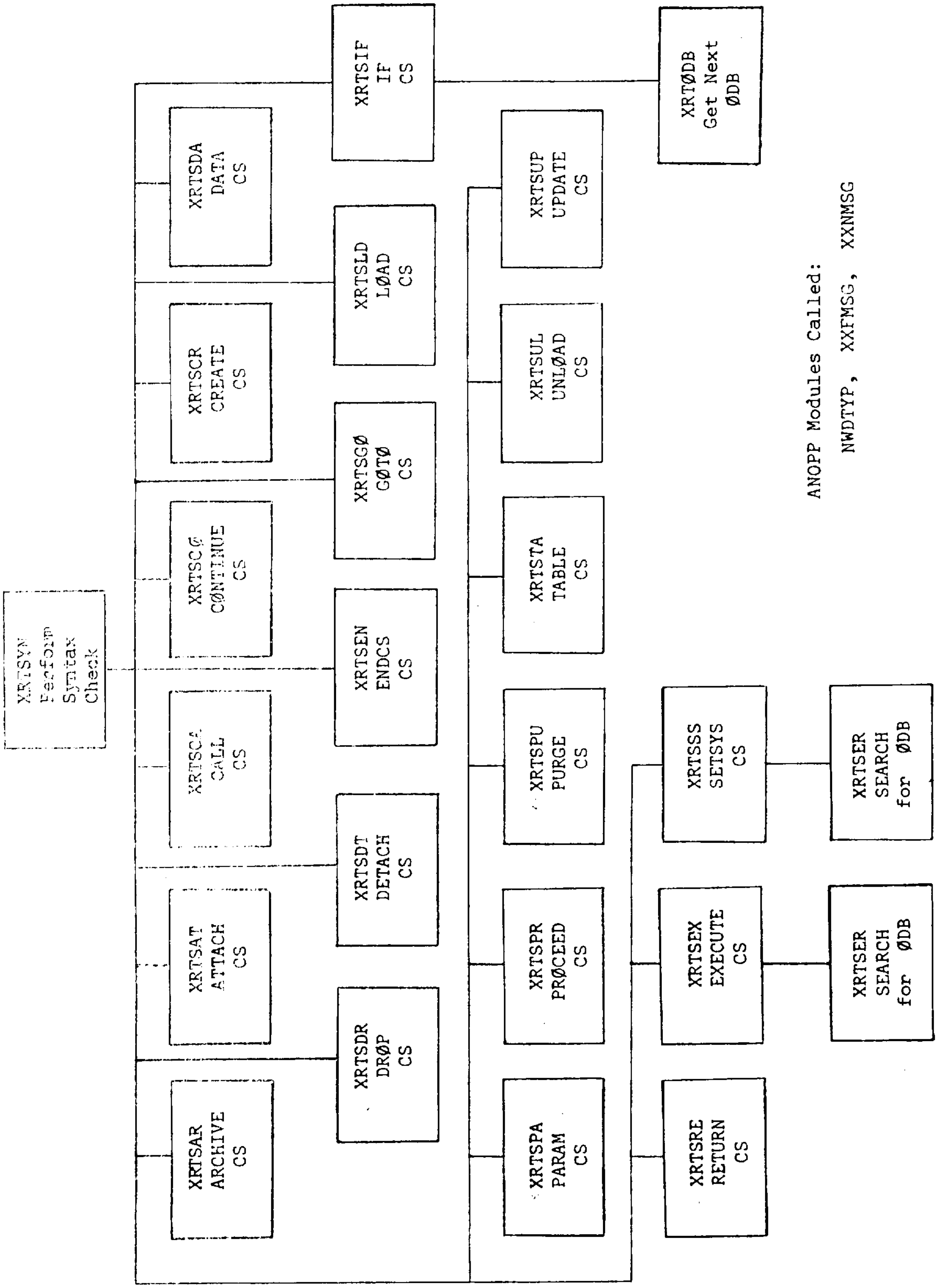
Figure 17. XRT Hierarchy Chart



ANOPP Modules Called:

DSMX, ICI, MMCLØS, MMØPWD, MMPUTR,
MMVUM, NWDTYP, XCR, XFETCH, XPK,
XT1AL, XUNPK, XXFMSG, XXNMSG

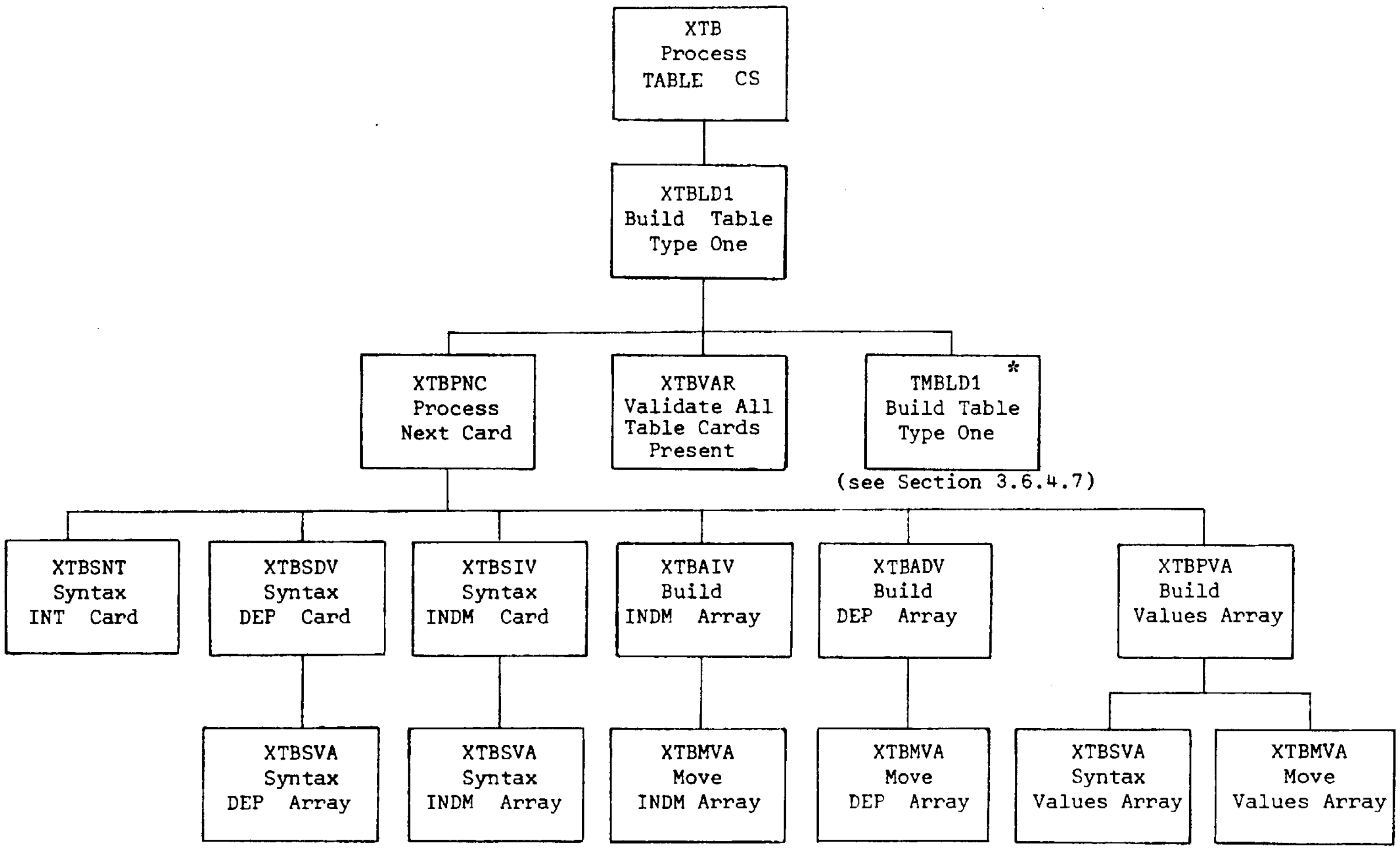
Figure 18. XRTCSS Hierarchy Chart



ANOPP Modules Called:

NWDTYP, XXFMSG, XXNMSG

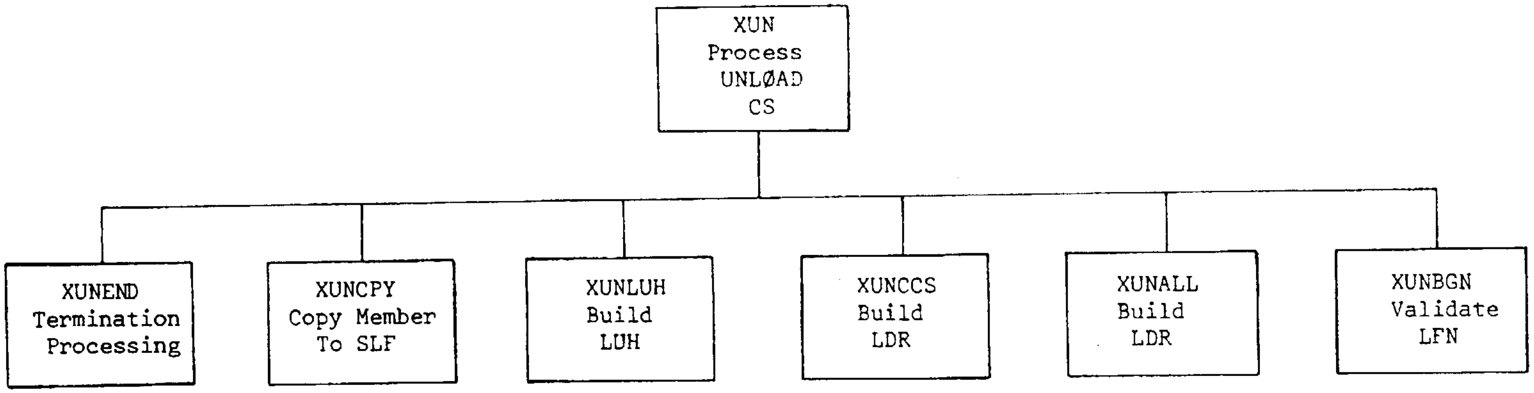
Figure 19. XRTSYN Hierarchy Chart



ANOPP Modules Called:

DSMF, DSMG, MMCLØS, MMGETR, MMØPRD,
NUMTYP, NWDTYP, TMERR, XCR

Figure 20. XTB Hierarchy Chart



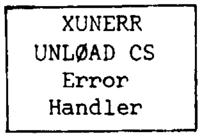
ANOPP Modules Called:

DSMF, DSMG, DSMQ, DSMX, IFETCH,
ILØC, MMCRMX, MMDØMC, MMIØMC, MMRMD,
MMSUD, XSØRTF, XTBDMP, XT1FV, XT3FVA

CDC

CLØSEM, ENDFILE, FILESQ, GET,
ØPENM, PUT, PUTP

Figure 21. XUN Hierarchy Chart



ANØPP Modules Called:

XEXIT, XFETCH, XPLINE, XTRACE

Figure 22. XUNERR Hierarchy Chart

ANOPP Modules Called:

RVALUE, XEXIT, XFETCH, XPLINE

Figure 23. XXFMSG Hierarchy Chart

ANOPP Modules Called:

XFETCH, XPLINE, XXFMSG

Figure 24. XXNMSG Hierarchy Chart

3.6 ANOPP DATA BASE MANAGEMENT

3.6.1 Overview

The ANOPP Data Base Manager (DBM) provides ANOPP executive and functional modules with a machine independent method of storing and retrieving data on sequential and direct access storage devices. The following features are provided:

1. Creation of new data units on direct access storage devices.
2. Accessing of existing data units on direct access storage devices.
3. Multi-data unit sequential library files for offline storage and retrieval of data units.
4. Direct and sequential access of data members.
5. Fixed format, variable format, and unformatted record types.
6. Full record, partial record, and sequential element within record reading and writing of data members.

The ANOPP DBM provides a hierarchial data structure having direct (based on the relative record position) and sequential accessing of logical records. These data relationships are visualized in Figure 1.

The highest level of the hierarchy is termed the "data base", which is defined as the universe of data for a particular ANOPP run. The universe is composed of named "data units" and encompasses all units referenced, even though units may be loaded, attached, created, and detached at various times during an ANOPP run.

The "data unit" is the next level of the hierarchy. It is the highest level that may be directly referenced through ANOPP DBM control statements and subroutine calls. A Data Unit is physically stored on direct access storage devices and is comprised of one or more named "data members". A data unit name must be unique within a particular segment of an ANOPP run.

Each "data member" is uniquely named within a data unit and is comprised of a set of logically related and organized records. Each member contains a format specification which defines the type and structure of the records.

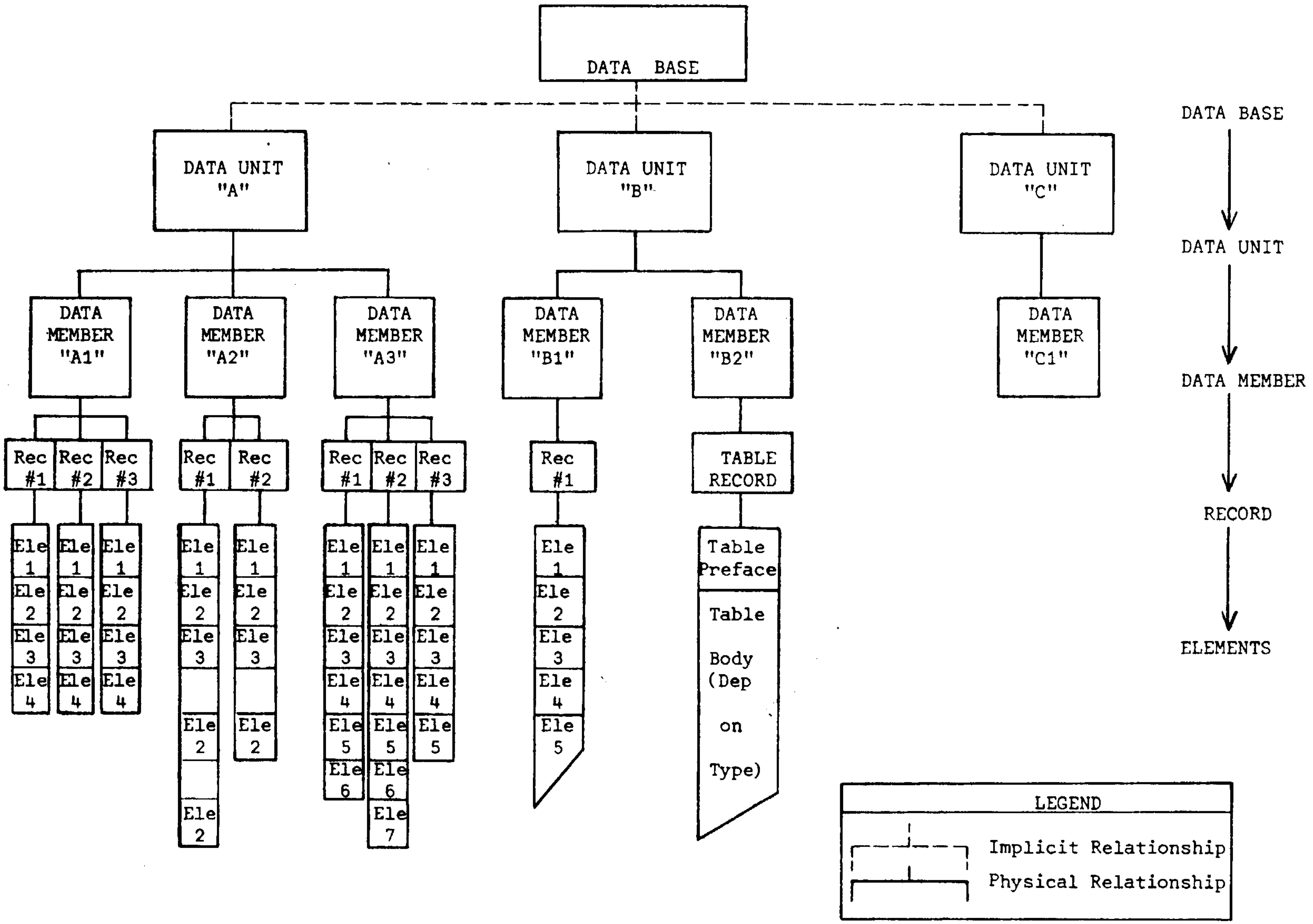


Figure 1. DBM Hierarchal Data Structure

Four record types are supported by the ANOPP DBM; fixed format, fixed format header with a variable number of fixed format trailers, card image, and unformatted records.

Depending on the type, records are comprised of one or more contiguous words or elements. On formatted members the format specification defines the type and length of individual elements (integer, real double precision, complex, character string, etc.) and thus the length in words of records. Unformatted members have variable length records whose lengths are defined as they are written.

3.6.2 DBM Control Statements

Several ANOPP DBM Control Statements have been defined to enable the ANOPP user to define a data base to the ANOPP Data Base Manager and to communicate the file names used by the external system to the DBM. In brief, the following control statements are featured:

LØAD	-	load data units from a sequential library;
UNLØAD	-	unload data units to a sequential library;
ATTACH	-	attach a data unit from the external system;
DETACH	-	detach a data unit from the internal system;
CREATE	-	create a new data unit;
ARCHIVE	-	permanently write-protect a data unit;
PURGE	-	detach a data unit internally and drop it from the external system
DRØP	-	release an external file name from the external system;
TABLE	-	build a table, to be accessed using Table Manager, on a data member.

A detailed description of each control statement is provided in Section 3.5.2.

3.6.3 Member Manager

3.6.3.1 General Description

The member manager as part of the ANOPP data management system provides basic open/

close, read/write, and position functions for module writers via calls to specific member manager routines. These routines are:

OPEN

MMØPWD	open member to write directly
MMØPWS	open member to write via scratch
MMØPRD	open member to read

PUT

MMPUTR	write a record
MMPUTW	write a partial record on n words
MMPUTE	write a partial record of n elements

GET

MMGETR	read a record
MMGETW	read a partial record of n words
MMGETE	read a partial record of n elements

CLOSE

MMCLØS	close a member

POSITION

MMSKIP	skip n records
MMREW	rewind member
MMPØSN	position member to record n

Data members are made accessible through calls to the member manager "open" routines. These routines establish and maintain the control structures required for maintaining multiple members on a single unit. The following access modes are provided:

1. Open for reading which allows for random and sequential retrieval of full and partial records;

2. Open for direct writing which enables direct storage of records to a data member on a unit; and

3. Open for scratch writing which provides storage of records to a data member on a scratch unit until the member is closed.

The number of members which may be concurrently open is limited by the amount of available Global Dynamic Storage and the number of allocated Data Unit Directory (DUD) entries. Each data unit which has one or more members open requires dynamic storage for a file table and buffer through which it interfaces with the computer operating system. Each open member also requires an Active Member Directory entry and a Member Control Block which are also in dynamic core. Additionally, each member opened for indirect (scratch) write uses a DUD entry with associated file table and buffer.

Three Member Manager PUT subprograms are provided which enable the user to write full and partial records to a data member that is open for direct or indirect write. Calls to these subprograms may be intermingled as required with only three limitations:

1. MMPUTE may not be used with unformatted members.
2. MMPUTW calls which precede MMPUTE calls must write the number of words required for MMPUTE to begin writing at the beginning of the next element.
3. MMPUTE and MMPUTW calls which immediately precede MMPUTR calls must end the record which they were writing.

A member on a unit can be simultaneously open to read and write. The old version is available for reading until the new version is completed and closed at which time all MM internal links to the old version are destroyed. The new version being written (in "scratch" or "direct" mode) does not physically replace the old version but instead is written at a different location on the unit. Therefore, in reading and writing simultaneously, no synchronization of PUTs and GETs is required. Any record on the old version is available for reading regardless of which record on the new version is being written. Since the open to write call is non-destructive, the open to read call may occur before or after the open to write call. The open to read call on this member (MMØPRD) should have a corresponding close call (MMCLØS) before the new version of the member is closed. If it does not, an informative message is issued. Furthermore, the NAME argument provided to the open to read, subsequent gets, and corresponding close calls must not have the same core location (i.e., same variable name) as the NAME argument provided to the open to write and subsequent puts and corresponding close calls.

Example (where UNIT1 (MEM1) is an existing member and not open currently):

```
DIMENSION NAMER(3), NAMEW(3)
NAMER(1)  =  5HUNIT1
NAMER(2)  =  4HMEM1
NAMEW(1)  =  5HUNIT1
NAMEW(2)  =  4HMEM1
CALL MMØPRD (NAMER)
CALL MMØPWD (NAMEW, other arguments)
CALL MMPUTR (NAMEW, other arguments)
CALL MMGETR (NAMER, other arguments)
CALL MMCLØS (NAMER)
CALL MMCLØS (NAMEW)
```

Three Member Manager Position subprograms are available which provide rewind, forward and reverse skip, and random record positioning. Usage of these routines requires that a member be open for read.

The close member module (MMCLØS) should be called for all open members prior to returning to ANOPP Executive control. If an executive module fails to close a member, the error will not be detected, subsequent use of that member will be inhibited, and unpredictable errors in subsequent executive and functional modules may occur. If a functional module fails to close a member, subprogram XFMMM will logically close it and issue an informative message. However, members which were open to write (and not closed) will not be entered or updated in the Data Member Directory for the particular data unit on which they were written. Thus, members which did not exist prior to being opened will be eliminated from the ANOPP universe of data and updated members will be restored to their pre-update condition.

3.6.3.2 Subroutine Arguments

Several subroutine arguments are common to more than one member manager subroutine.

NAME - a three word array, the first 2 words specifying the unit and member names respectively. Each name is 1-8 alphanumeric characters beginning with an alphabetic character. The third word is reserved for the MM and must not be altered by the user.

FORMAT - specifies the format of the records which will be created. It is given as a character string terminated with a $. The acceptable codes for elements which comprise the record include:

I:	integer (one word)
RS:	real single precision (floating point-one word)
RD:	real double precision (floating point-two words)
CS:	complex single precision (floating point-two words)
L:	logical (one word)
CD:	complex double precision (floating point-four words)
Ai:	character string of i characters which is assumed to be packed 8 characters per word, $1 \leq i \leq 132$.
$:	format string terminator character.

Each element code is separated by a comma. A multiplier may optionally precede parentheses enclosing a single element or a group of elements. The multiplier specifies the number of times the element(s) are to be repeated. The character * may be used as an indefinite multiplier when preceding the last element(s) of the format. The * specifies an indefinite repeat of the element group. There are four types of formats:

UNFØRMATTED - the records are undefined format and variable length. A zero is coded for FØRMAT.

FIXED LENGTH FORMAT - The format does not contain the indefinite multiplier (*). Each record will be of the same length determined by the format.

VARIABLE LENGTH FORMAT - The format includes the indefinite multiplier (*). The records are variable length depending on the number of elements written which may or may not include the indefinite repeat group. However, partial repeat groups may not be written.

CARD IMAGE FORMAT - The format consists of "CI" and will be interpreted as "10A8$". It is, therefore, a special purpose fixed format member.

For example:

FORMAT = 6H10 I $
specifies fixed length formatted records of 10 words (10 integer elements).

FORMAT = 19HI,2 A10, 2 (I,RS) $
specifies a fixed length formatted record of 9 words (7 elements) as follows: integer, 10 characters (2 words), 10 characters (2 words), integer, real single precision, integer, real single precision.

FORMAT = 15H2(I), *(I,CS) $
specifies a variable length formatted record of two integers followed by repeating group of 2 elements (3 words) of integer and complex single precision. The number of elements on a record may be 2, 4, 6, 8, etc., depending on the number of repeat groups actually written.

FORMAT = 0
then the records will be variable length with undefined format.

FORMAT = 2HCI
then the records are fixed format card images.

MNR - specifies the maximum number of records to be output to a data member. If MNR is zero a default value of 10,000 records is used. The MNR argument is used by Data Member Manager in computing the size of Record Directory blocks which are used as indices in subsequent random and sequential record retrieval. An attempt to write more than MNR data records to a particular member will result in immediate termination of the ANOPP run.

STATUS - conditions, which are of interest to a user of Member Manager, are returned in this argument by open, get, and positioning subroutines. If STATUS is:

0	conditions are normal for the operation involved,
-1	Member Manager is currently positioned in the midst of a record or the just executed MMGETR transferred a partial record,
-2	End of record occurred on a partial get,
-3	End of member was detected on the last get or position operation,
-4	Beginning of member was detected on the last position operation,
-5	The data unit specified in the last open member request does not exist, and
-6	The data member specified in the last open member request does not exist.

3.6.3.3 Open Data Member Subroutines

3.6.3.3.1 MMØPRD - Open for Read

Purpose: MMØPRD makes an existing data member available for subsequent random and sequential accessing of data.

Format: CALL MMØPRD (NAME, IHDR, STATUS)

Arguments:

NAME - a three word array containing the names of the data unit and member to be opened. Upon returning, the names are unchanged and the third word contains the integer IDX to the Member Control Block.

IHDR - a two word array which on return contains, in the first word, the length of the largest data record and, in the second word, the number of data records written on the data member.

STATUS - an integer less than or equal to zero is returned. If zero is returned, the data member was opened. A negative status indicates that the data member is not open.

Description: The initial steps in opening a data unit are validation of the name argument and determining if the data member is in use via Data Table Manager. The name validation routine (MMVNM) fetches alternate names for both data unit and member, edits them for proper form (1 to 8 character alphanumeric with leading alphabetic character), and attempts to locate the named data unit in the Data Unit Directory (DUD). If a DUD entry is not found, a status of -5 is set and the open routine returns to its caller. If a DUD entry is found, a routine (MMVTD) is called to determine if the data member is also open to Data Table Manager (DTM). If it is, ANOPP execution is terminated immediately with an appropriate message.

If the data member is not open to DTM, subprogram MMIØMC is then called to increment the DUD entry's open member count and insure that the data unit is open. The data unit's Data Member Directory (DMD) is then read into core and searched for the named data member. If an entry for the data member is not found a -6 status is set, the open member count is decremented and if it is zero the data unit is closed and MMØPRD returns to its caller. When an entry is found, however, an Active Member Directory entry and a Member Control Block are established for the data member; the maximum record length and number of user records are retrieved from the data member's Data Member Header; and a status of zero is returned to the user.

Error Conditions: MMØPRD prints an informative message and returns a non-zero status to the caller if either the unit named in the open request is not in the unit directory or the member named is not in the member directory.

MMØPRD aborts with a message if the member is already open to read, if the member is an active data table, or if an attempt to expand the member control block is unsuccessful.

3.6.3.3.2 MMØPWD - Open for Direct Write

Purpose: MMØPWD makes a data member available for subsequent sequential output directly to its data unit.

Format: CALL MMØPWD (NAME, FØRMAT, MNR, STATUS)

Arguments:

NAME - a three word array containing the names of the data unit and member to be opened. Upon returning, the names are unchanged and the third word contains the integer IDX to the Member Control Block.

FØRMAT - a Hollerith literal specifying the number and types of data elements in each data record. Legal values are discussed in Subsection 3.6.3.2.

MNR - an integer number, greater than or equal to zero, which specifies the maximum number of data records that may be written to the data member.

STATUS - a negative or zero integer is returned indicating, if zero, that the data member is open, or, if negative, not open.

Description: The initial steps in opening a data member for direct output are validating the name argument and determining if the data member is in use via Data Table Manager. The name validation routine (MMVNM) fetches alternate names for both data unit and member, edits them for proper form (1 to 8 character alphanumeric with leading alphabetic character), and attempts to locate the named data unit in the Data Unit Directory (DUD). If a DUD entry is not found, a status of -5 is set and the open routine returns to its caller. If a DUD entry is found, a routine (MMVTD) is called to determine if the data member is also open to Data Table Manager (DTM). If it is, ANOPP execution is terminated immediately with an appropriate message. If the data member is not open to DTM, the data unit's direct write flag is checked to determine if another data member is open for direct writing on the data unit. If the direct write flag is set, ANOPP execution is terminated with an appropriate message. Otherwise, an entry is made in the Active Member Directory, the data unit's direct write flag is set, a Member Control Block is built containing the

Data Member Header, and the open member count in the data unit's DUD entry is incremented (via MMIØMC) thus insuring that the data unit is open.

Error Conditions: MMØPWD prints an informative message and returns a non-zero status to the caller if the unit named in the open request is not in the unit directory.

MMØPWD aborts with a message if the data unit has been archived or is already open for direct write.

3.6.3.3.3 MMØPWS - Open for Indirect Write

Purpose: MMØPWS makes a data member available for sequential output to a scratch data unit. When closed, the data member is copied to the data unit named in the open.

Format: CALL MMØPWS (NAME, FØRMAT, MNR, STATUS)

Arguments:

NAME - a three word array containing the names of the data unit and member to be opened. Upon returning, the names are unchanged and the third word contains the integer IDX to the Member Control Block.

FØRMAT - a Hollerith literal specifying the number and types of data elements in each data record. Legal values are discussed in Subsection 3.6.3.2.

MNR - an integer number, greater than or equal to zero, which specifies the maximum number of data records that may be written to the data member.

STATUS - a negative or zero integer is returned indicating, if zero, that the data member is open or, if negative, not open.

Description: The initial steps in opening a data unit are validating the name argument and determining if the data member is in use via Data Table Manager. The name validation routine (MMVNM) fetches alternate names for both data unit and member, edits them for proper form (1 to 8 alphanumeric with leading alphabetic character), and attempts to locate the named data unit in the Data Unit Directory (DUD). If a DUD entry is not found, a status of -5 is set and the open routine returns to its caller. If a DUD entry is found, a routine (MMVTD) is called to determine if the data member is also open to Data Table Manager. If it is, ANOPP execution is terminated immediately with an appropriate message. If the data member is not open to DTM, a scratch data unit is created, an Active Member Directory entry is built, a Member Control Block is built containing the Data Member Header, and the open member count is incremented to open both the actual and scratch data units.

Error Conditions: MMØPWS prints an informative message and returns a non-zero status to the caller if the unit named in the open request is not in the unit directory.

MMØPWS aborts with a message if the data unit is archived.

3.6.3.4 Put Subroutines

3.6.3.4.1 MMPUTR - Put Record

Purpose: MMPUTR writes a complete record to a named data unit.

Format: CALL MMPUTR (NAME, ARRAY, NWDS)

Arguments:

NAME - a three word array which specifies a data member, opened to write, on which the record is to be written.

ARRAY - an array containing the record to be written to the data member.

NWDS - length, greater than or equal to zero, of the record to be written.

Description: There are three initial validations performed by subprogram MMPUTR. First, a call to subprogram MMEDNM insures that the NAME argument used in calling MMPUTR is the same as that used to open the member and that the member is open for write. Second, the Member Control Block (MCB) is checked to determine if the previous record was completed. If is was not, ANOPP execution is terminated and a message specifying the cause is output. Third, if the previous record was completed, the NWDS argument must be zero or positive. If NWDS is negative, ANOPP execution is terminated.

The next level of validation is dependent on the data member's format type. For unformatted members, no further validation is performed. For fixed format, the NWDS argument must equal the fixed record length from the MCB. And for variable format records, NWDS must be equal to the length of the fixed part of the record plus an integral (or zero) number of fixed length trailers.

Records are put to the member using subprogram MMPUT which builds the Record Directory and maintains the control information in the MCB.

Error Conditions: MMPUTR aborts with a message if the NAME argument is invalid, if the previous record is incomplete, if the NWDS argument is negative, or if the record to be written does not end on a legitimate record format boundary.

3.6.3.4.2 MMPUTW - Put Partial Record Words

Purpose: MMPUTW writes a partial record of a specified number of words to a fixed format, variable format, or unformatted data member.

Format: CALL MMPUTW (NAME, ARRAY, NWDS, EØR)

Arguments:

NAME - a three word array which specifies a data member, opened to write, on which the partial record is to be written.

ARRAY - an array containing partial record to be written.

NWDS - number of words, greater than or equal to zero, to be written from ARRAY.

EØR - logical end-of-record flag -- .TRUE. terminates the record and .FALSE. record is to be left open for additional partial puts.

Description: MMPUTW performs two initial validations. First, subprogram MMEDNM insures that the NAME argument is valid for the put operation. Then the NWDS argument must not be negative. If either of these validations fails, ANOPP is terminated and a message describing the error is output.

Final validation of a record length is performed when the EØR argument is true and the data member is formatted. If the data member is fixed format the total record length must equal the record length implied by the format. If variable format, the total record length must equal the length of the fixed part of the record plus the length of an integral (or zero) number of fixed length trailers.

Finally, subprogram MMPUT writes the partial record to the member, updates the control information in the MCB, and, when EØR is true, updates the Record Directory.

Error Conditions: MMPUTW aborts with a message if the NAME argument is invalid, if the record length is incompatible with the format, or if the number of words argument is negative.

3.6.3.4.3 MMPUTE - Put Partial Record Elements

Purpose: MMPUTE writes a partial record of a specified number of elements to a formatted data member.

Format: CALL MMPUTE (NAME, ARRAY, NEL, EØR)

Arguments:

NAME - a three word array which specifies a data member, opened to write, on which the elements are to be written.

ARRAY - an array containing the partial record to be written.

NEL - number of elements, greater than or equal to zero, in ARRAY.

EØR - logical end-of-record flag. .TRUE. terminates the record; .FALSE. record is to be left open for additional partial put requests.

Description: MMPUTE validates the NAME argument using subprogram MMEDNM to insure that the data member is open to write. It then checks the format type since elements may not be put to an unformatted member.

Next the NEL argument is checked to insure that it is not negative; the Format Specification Table (FST) index in the Member Control Block (MCB) is set by MMSFEI based on the number of words already put to the current record (also in the MCB); and the number of words required to put NEL elements to the data member is determined using MMGNWE. If the end-of-record flag (EØR) is true, the total record length, including NEL elements, is checked against the format and if it is not valid, a message is issued and ANOPP is terminated. Finally, if all validations are passed, NEL elements are written to the member via subprogram MMPUT.

Error Conditions: MMPUTE aborts with a message if the NAME argument is invalid, if the record type is unformatted (improper use of this call), if the number of elements in the array containing the partial record to be written is negative, if the record length is incompatible with the format, or if the total record length exceeds the fixed format.

3.6.3.5 Get Subroutines

3.6.3.5.1 MMGETR - Get Record

Purpose: MMGETR attempts to read a complete record from a named data member.

Format: CALL MMGETR (NAME, ARRAY, MAXWDS, NWDS, STATUS)

Arguments:

NAME - a three word array which specifies a data member, opened for read, from which the record is to be read.

ARRAY - an array into which the data record is to be read.

MAXWDS - maximum number of words which may be read into ARRAY (i.e., the assumed length of ARRAY) must be greater than zero.

NWDS - returned by MMGETR, integer number of words actually read into ARRAY. NWDS will be less than or equal to MAXWDS.

STATUS - returned by MMGETR, integer status of the read operation:
0 - a complete record was read. NWDS is less than or equal to MAXWDS.
-1 - the record in ARRAY was truncated due to lack of room, NWDS equals MAXWDS (record length is greater than MAXWDS) and member is positioned to the beginning of the next record.
-3 - the end-of-member was detected, NWDS equals zero.

Description: MMGETR validates the NAME argument using subprogram MMEDNM to insure that the data member is open for reading. The MAXWDS argument is checked to insure that it is greater than zero. The Member Control Block (MCB) is then checked to determine if the member is positioned within a record. If it is the member is repositioned to the beginning of the next record. Subprogram MMGET is then called to read MAXWDS words into ARRAY. MMGET returns a status of -1 if the record length is greater than MAXWDS words, -2 if a full record was read, and -3 if the end of member was detected. MMGETR changes a -2 status to zero and returns the status to the user.

Error Conditions: MMGETR aborts with a message if the NAME argument is invalid or if the maximum number of words which may be read into ARRAY is less than or equal to zero.

3.6.3.5.2 MMGETW - Get Partial Record - Words

Purpose: MMGETW reads a partial record of a specified number of computer words from a data member.

Format: CALL MMGETW (NAME, ARRAY, NWR, NWDS, STATUS)

Arguments:

NAME - a three word array which specifies a data member, opened for read, from which the partial record is to be read.

ARRAY - an array into which the partial record is to be read.

NWR - number of words to be read into ARRAY, must be greater than zero.

NWDS - number of words actually read into ARRAY, returned by MMGETW; NWDS will be less than or equal to NWR.

STATUS - integer status of the read operation, returned by MMGETW:
 0 - a partial record of NWR words was read, NWDS equals NWR;
-2 - a partial record of NWDS words was read ending the record, NWDS is less than or equal to NWR;
-3 - end-of-member was detected on the read, NWDS equals zero.

Description: MMGETW validates the NAME argument using subprogram MMEDNM to insure that the data member is open for reading. The NWR arguments is checked to insure that it is greater than zero. NWR words are then read from the current position of the data member by subprogram MMGET. MMGET returns a status of -1 if NWR words were read, -2 if an end-of-record was detected, and -3 for end-of-member. A status of -1 is changed to zero prior to returning to the user.

Error Conditions: MMGETW aborts with a message if the NAME argument is invalid or if the number of words to be read is less than or equal to zero.

3.6.3.5.3 MMGETE - Get Partial Record - Elements

Purpose: MMGETE reads a partial record of a specified number of elements from a formatted data member.

Format: CALL MMGETE (NAME, ARRAY, MAXWDS, NER, NEL, STATUS)

Arguments:

NAME - a three word array which specifies a data member, opened for read, from which the partial record is to be read.

ARRAY - an array into which the partial record is to be read.

MAXWDS - maximum number of words which may be read into ARRAY (i.e., the assumed length of ARRAY) must be greater than zero.

NER - number of elements to be read into ARRAY, must be greater than zero.

NEL - number of elements actually read into array (returned by MMGETE), will be greater than or equal to zero.

STATUS - integer status of the read operation, returned by MMGETE:
 0 - a partial record of NER elements was read, NEL equals NER:
-1 - a partial record of NEL elements was read, NEL is less than NER;
-2 - a partial record of NEL elements was read ending the record, NEL is less than or equal to NER;
-3 - end-of-member was detected on the read, NEL is zero.

Description: MMGETE validates the NAME argument using subprogram MMEDNM to insure that the data member is open for reading. The length of ARRAY (MAXWDS) and number of elements to be read (NER) are then edited for greater than zero and the format type of the member is checked to insure that it is formatted. If an error is found, ANOPP is terminated with an appropriate message. The Format Specification Table (FST) index is then set

ORIGINAL PAGE IS
OF POOR QUALITY

by subprogram MMSFEI based on the number of words previously read from the current record. Subprogram MMGNWE is then called to obtain the number (NEL) and combined length (NWDS) of consecutive elements, up to a maximum of NER elements, whose combined length does not exceed MAXWDS words. NWDS words are then read from the member using subprogram MMGET. MMGET returns the STATUS and number of words actually read (NWR). If NWR is less than NWDS, the number of elements read (NEL) is obtained from subprogram MMGNEW. Then, if the STATUS does not equal -2 and NEL equals NER, STATUS is set to zero.

Error Conditions: MMGETE aborts with a message if the NAME argument is invalid, the length of the record array is not greater than zero, the number of elements to read is not greater than zero, the data member is unformatted (misuse of this call), or if the get of a formatted record does not end on an element boundary.

3.6.3.6 Position Subroutines

Data Member Manager provides the following capabilities for positioning within a data member that is open for reading:

1. Position to a Specified Record
2. Position to the beginning of the Current Record
3. Position to the beginning of the Data Member
4. Position forward or backward a Specified Number of Records.

3.6.3.6.1 MMPØSN - Position to a Specified Record

Purpose: MMPØSN positions a data member to the beginning of a data record specified by its numeric sequence on the member.

Format: CALL MMPØSN (NAME, NREC, STATUS)

Arguments:

NAME - a three word array specifying the data member, opened for read, which is to be positioned.

NREC - integer number specifying the record to which the data member is to be positioned.

STATUS - integer status of the position operation:
 0 - data member is positioned to the specified record;
-3 - data member is at the end of the member;
-4 - data member is at the beginning of the member.

Description: MMPØSN edits the NAME argument using subprogram MMEDNM to determine if the data member is open to read. If it is not, ANOPP is terminated with an appropriate message. The internal record position information in the Member Control Block (MDB) is reset to the beginning of the record and the current record number (CRN) in the MCB is set equal to the NREC argument. If NREC is greater than the number of records written to the member, the CRN in the MCB is set to number of records written plus one. If NREC is negative or equal to zero, then CRN is set to one.

Error Conditions: MMPØSN aborts with a message if the NAME argument is invalid.

3.6.3.6.2 MMREW - Rewind Data Member

Purpose: MMREW positions a data member to the beginning of the first data record on the member.

Format: CALL MMREW (NAME)

Arguments:

NAME - a three word array specifying the data member, opened for reading, which is to be positioned.

Description: MMREW edits the NAME argument using subprogram MMEDNM to determine if the data member is open to read. If it is not ANOPP is terminated with an appropriate message. The internal record position information is then set to the beginning of the record and the current record number is set to one.

Error Conditions: MMREW aborts with a message if the NAME argument is invalid.

3.6.3.6.3 MMSKIP - Skip Records

Purpose: MMSKIP positions a data member forward or backward by a specified number of records.

Format: CALL MMSKIP (NAME, NREC, STATUS)

Arguments:

NAME - a three word array specifying the data member, opened for reading, which is to be positioned.

NREC - integer number of records to be skipped:
if negative - skip backward NREC records;
if positive - skip forward NREC records;
if zero - position to the beginning of the current record.

STATUS - integer status of the position operation:
0 - data member is positioned to the specified record;
-3 - data member is at the end of the member;
-4 - data member is at the beginning of the member.

Description: MMSKIP edits the NAME argument using subprogram MMEDNM to insure that the data member is open for reading. If it is not, ANOPP as terminated and an appropriate message is issued. The internal record position information is set to the beginning of the current record and NREC is added to the current record number in the Member Control Block (MCB). If the current record number is now negative, it is set to one and the STATUS argument is set to -4. If the current record number is greater than the number of records available on the member, it is set to the number of available records plus one and the STATUS argument is set to -3. This provides an end-of-member condition on a subsequent read.

Error Conditions: MMSKIP aborts with a message if the NAME argument is invalid.

3.6.3.7 MMCLØS - Close Data Member Subroutine

Purpose: MMCLØS closes a previously open data member making it unavailable for subsequent access.

Format: CALL MMCLØS (NAME)

Arguments:

NAME - the three word array identifying the data unit and member that was used in opening the data member.

Description: MMCLØS edits the NAME argument using subprogram MMEDNM. If the data member is not open, processing is terminated. If the member was open for reading, it is logically closed using subprogram MMCLSE. If it was open for direct write the Data Member Directory (DMD) is updated, the Data Member Header (DMH) is written to the data unit, the direct write flags in the Data Unit Directory (DUD) and Active Member Directory (AMD) entries are cleared, and the member is logically closed using MMCLSE. In all cases, when MMCLSE is called the open member count is decremented and when the open member count equals zero the unit is logically closed.

The closing of a data member that is open for indirect write is a little more complex. First, the data unit may not have another data member open for direct write or ANOPP will be terminated. Second, the DMD and DMH must be updated and written on the scratch data unit that was created when the data member was opened. Third, the data member on the scratch unit is opened for read so it can be copied, using Data Member Manager, to the actual named data unit. The open member count on the scratch unit is set to 1 and the direct write flag is cleared. Fourth, a record buffer is requested in Global Dynamic core to be used in copying the member. Fifth, the Member Control Block created when the member was opened to write is modified to permit output directly to the data unit named in the open call. Sixth, the member is copied from the scratch unit to the actual unit, and the scratch unit is closed and discarded. Finally, the DMD and DMH are updated and written to the actual data unit, the member is logically closed via MMCLSE, and the dynamic core used in the copy operation is freed.

Error Condition: MMCLØS aborts with a message if the NAME argument is invalid, if close write requested and member also open to read, if member was open to write direct and another member is open to write direct on the same data unit, or if insufficient Global Dynamic Storage is available for MMCLØS scratch copy.

3.6.3.8 Auxiliary Modules

An auxiliary module performs a function common to several member manager modules and is available for use by these modules.

3.6.3.8.1 MMERR - Member Manager Error Message Writer

Subroutine MMERR (NUM, NAM1, NAM2) processes fatal and non-fatal errors for the member manager modules and the DBM control statements ARCHIVE (XAR), ATTACH (XAT), CREATE (XCT), DETACH (XDT), and PURGE (XPU). NUM, the integer number of the error message to be printed, is negative if the error is fatal and positive if the error is non-fatal. TMERR prints the informative error message (indicated by the absolute value of NUM) with specific values involved in the error condition (indicated by input values NAM1 and NAM2). If the error is fatal, ANOPP is aborted by a call to XEXIT. If the error is non-fatal, a trace-back is performed to the major DBM module called by the user.

3.6.3.8.2 MMVUM - Validate Data Unit and Member

Purpose: MMVUM determines if a data unit and member are available in the present ANOPP operating environment.

Format: I = MMVUM (NAME)

Arguments:

NAME - a two word array containing the name of the data member and name of the unit on which it resides.

MMVUM - returns the following integer values:
-1 - data unit does not exist;
0 - both data unit and member exist;
1 - data member does not exist.

Description: MMVUM performs the same types of validations as subprogram MMØPRD; however, no informative messages are issued. The form of the data unit and member names is validated using subprogram XVNAME to insure that they are proper alphanumeric names. If either name is malformed an appropriate message is issued and ANOPP is terminated.

Alternate names for both data unit and member are fetched and the Data Unit Directory is searched for the named unit. If it is not found, a function value of -1 is returned to the caller. If the data unit is found, its Data Member Directory is read into core and searched for the named data member. If it is found, a function value of zero is returned. Otherwise, the function's value will be 1.

3.6.3.8.3 LØAD Control Statement Error Message Writer - XLDERR

Subroutine XLDERR(NUM, NAME, IVAL, IRAY, L) processes non-fatal errors for the DBM control statement LØAD.

3.6.3.8.4 UNLØAD Control Statement Error Message Writer - XUNERR

Subroutine XUNERR(NUM, NAME, IVAL, IRAY, L) processes non-fatal errors for the DBM control statement UNLØAD.

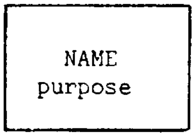
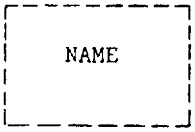
3.6.3.9 Hierarchy Charts

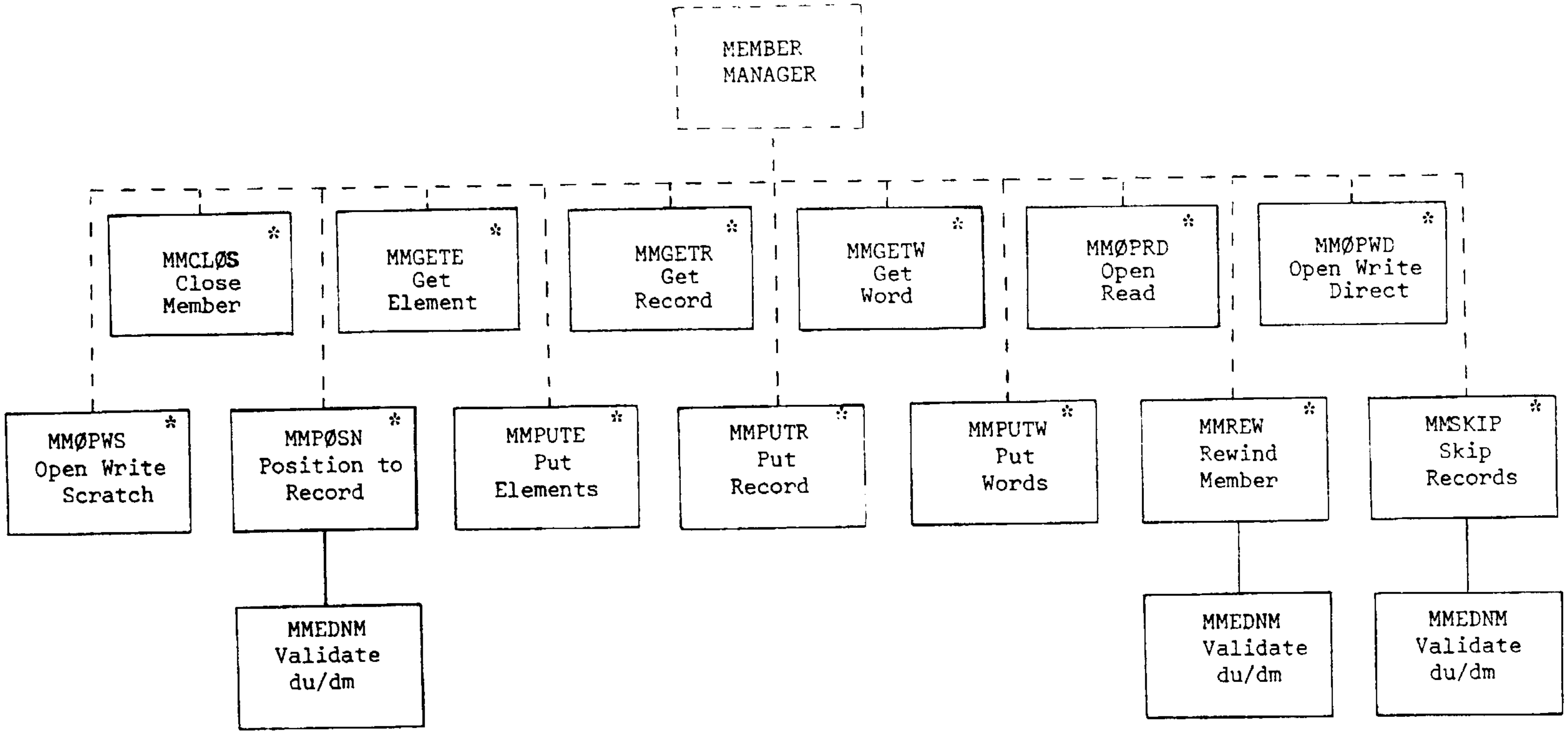
A hierarchy chart is a graphical representation of the logical relationship between modules. Figures 1-23 are the hierarchy charts for the member manager modules and the auxiliary modules.

In general, only member manager modules appear as a block entity in the charts and all member manager modules appear at least once. The charts are in alphabetical order with respect to module name except for Figure 1, which represents the logical grouping of the member manager modules. A hierarchy for the auxiliary modules are also among the alphabetical charts.

A module which is not part of member manager but is called by a member manager module is not shown as a block entity but is listed at the bottom of the chart. The module may be an ANOPP executive module which is part of the Executive Management System, the Dynamic Storage Management System, or the General Utilities. It may also be a subprogram provided by one of the CDC operating system libraries. In either case, the module is generally of a service or utility nature and may be called many times by various member manager modules. One of these service type modules may be of sufficient design purpose to the calling MM module that it should receive more emphasis than simply being listed. In these cases, the non-MM module is represented as a block entity for logical emphasis and is noted as such on the chart.

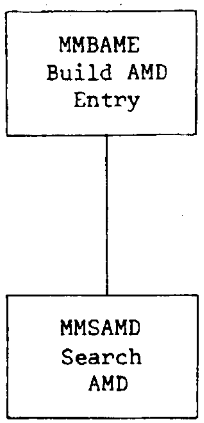
Symbols and headings used in the hierarchy charts are given below:

	NAME - module name purpose - brief description
	Represents logical module not existing as entity. It is used for logical groupings.
————————	indicates lower module is called by the higher module.
----------	implies logical grouping with no direct relationship
*	in upper right corner of module block indicates module is expanded as a separate entity.
ANOPP Modules Called:	a list of DBM, DSM, and General Utility modules called by the modules in this figure.
CDC System Library Subprograms Called:	a list of subprograms called by the modules in this figure and which are not part of ANOPP but are provided by CDC NOS operating system libraries.



ANOPP Modules Called:
ILØC, MMERR, XFAN

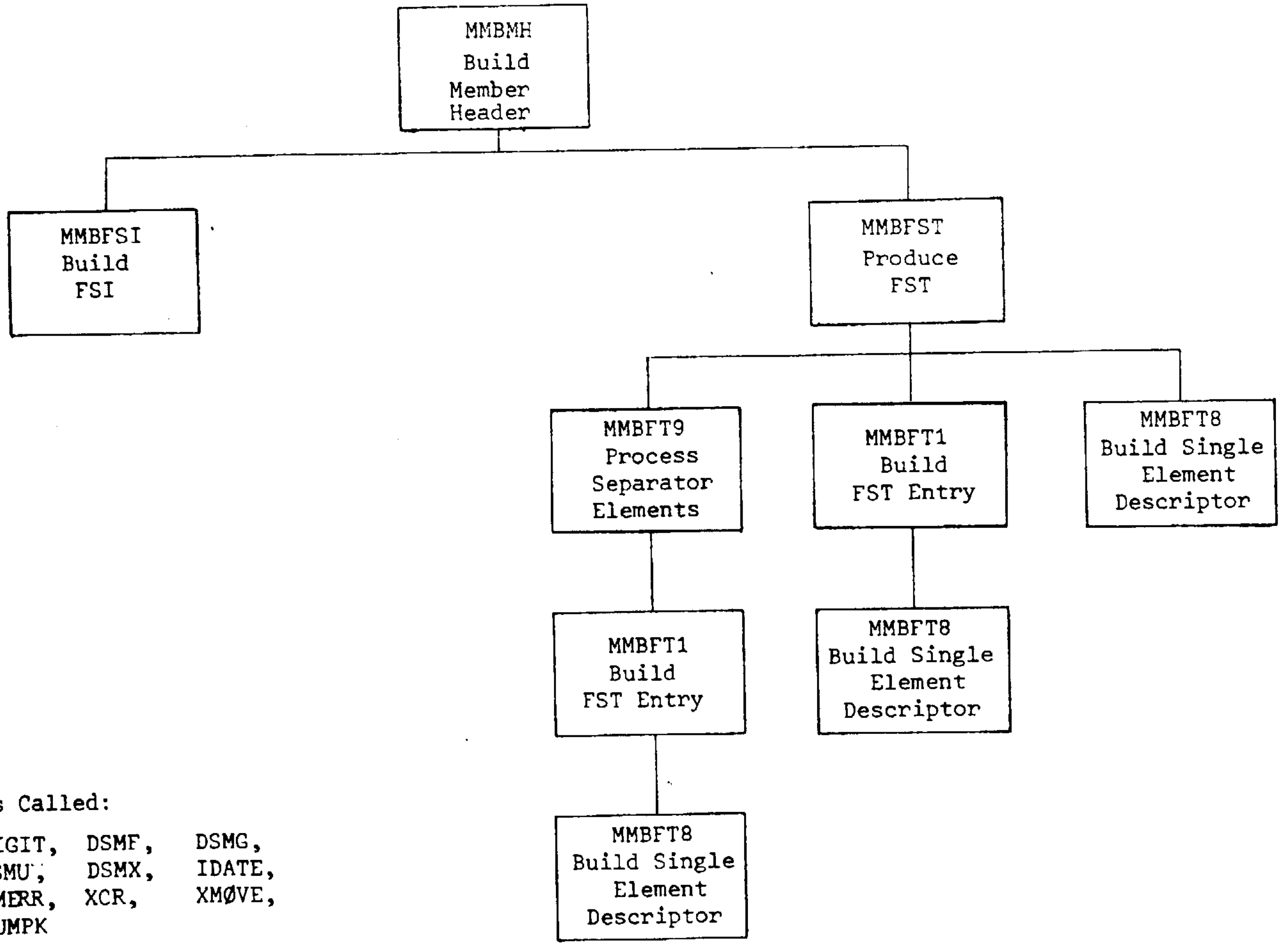
Figure 1. Member Manager Hierarchy Charts.



ANOPP Modules Called:

DSMX, ILØC, MMERR, XT3FV, XT3LK

Figure 2. MMBAME Hierarchy Chart

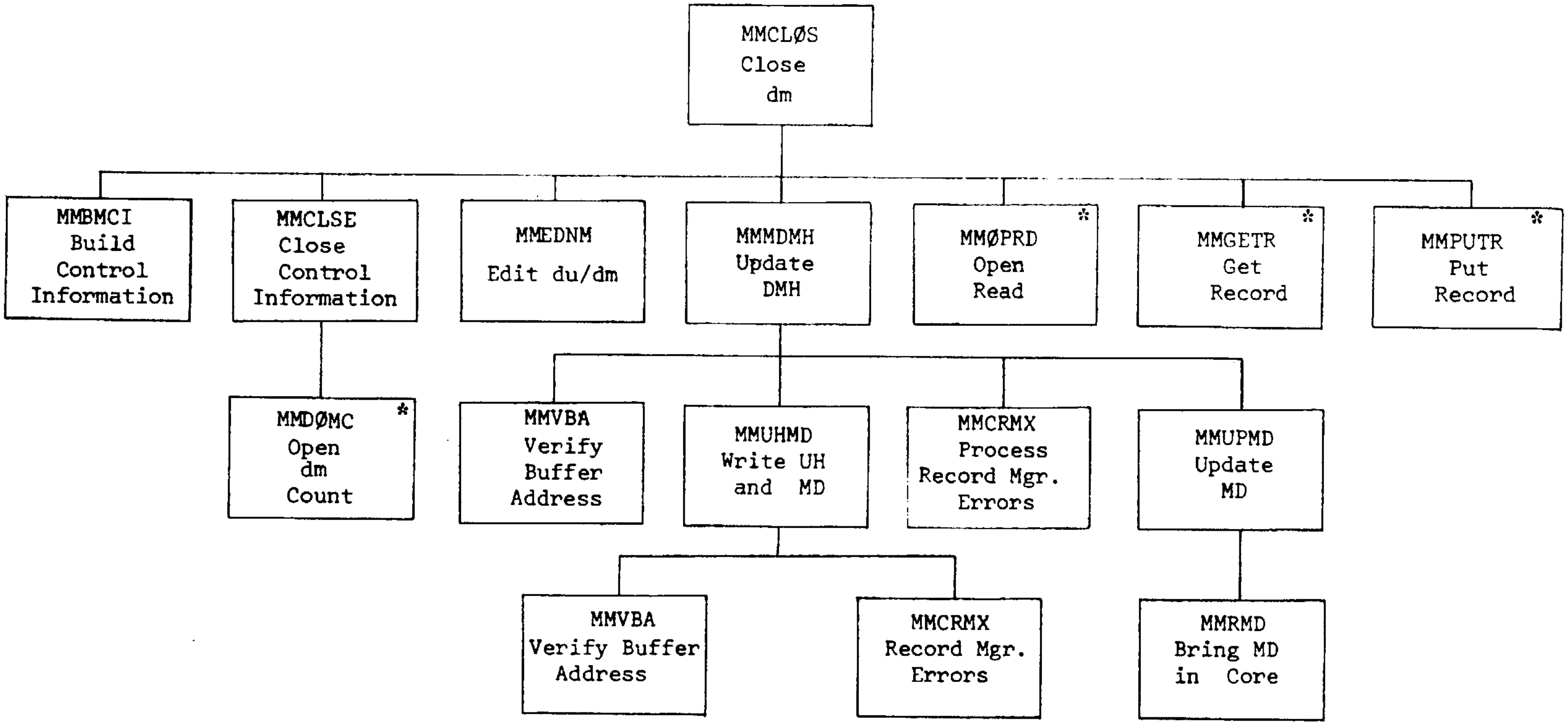


ANOPP Modules Called:

ALPHA, DIGIT, DSMF, DSMG,
DSML, DSMU, DSMX, IDATE,
ITIME, MMERR, XCR, XMØVE,
XPK, XUMPK

CDC System Library Subprograms Called:
SQRT

Figure 3. MMBMH Hierarchy Chart



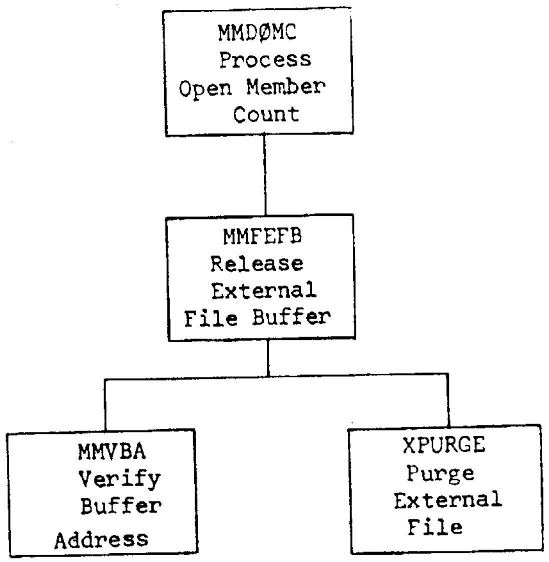
ANOPP Modules Called:

DSMF, DSMG, DSMX, ILØC, MMERR,
XFAN, XT1FV, XT3FV, XT3LK

CDC System Library Subprograms Called:

AND, GET, IFETCH, MASK,
ØR, PUT, REMARK, SHIFT

Figure 4. MMCLØS Hierarchy Chart



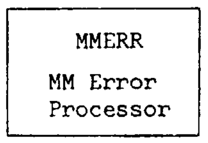
ANOPP Modules Called:

MMERR, DSMF, ILØC

CDC System Library Subprograms Called:

CLØSEM

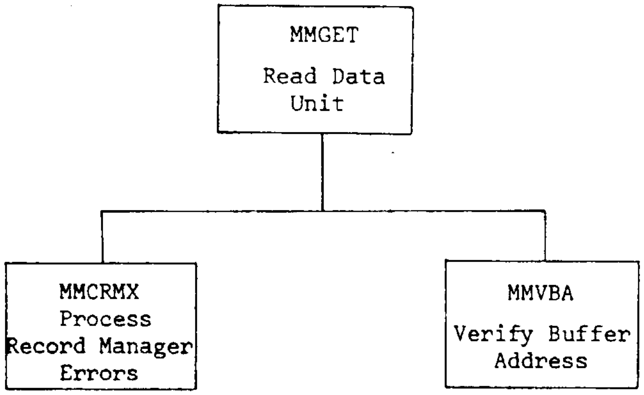
Figure 5. MMDØMC Hierarchy Chart



ANOPP Modules Called:

XEXIT, XFETCH, XPLINE,
XTRACE

Figure 6. MMERR Hierarchy Chart



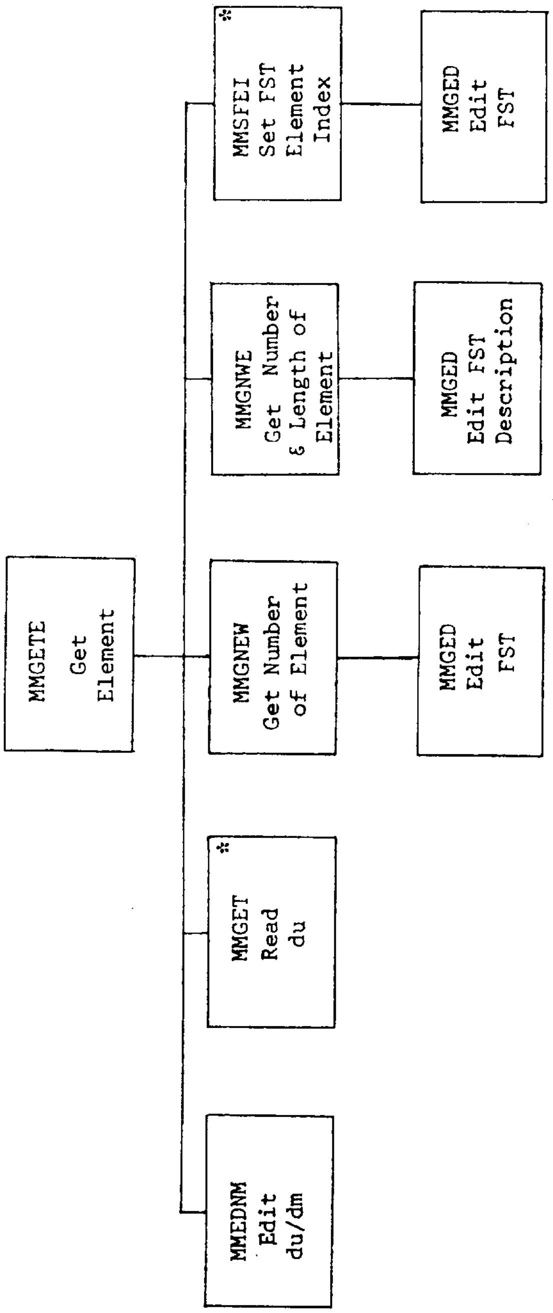
ANOPP Modules Called:

MMERR

CDC System Library Subprograms Called:

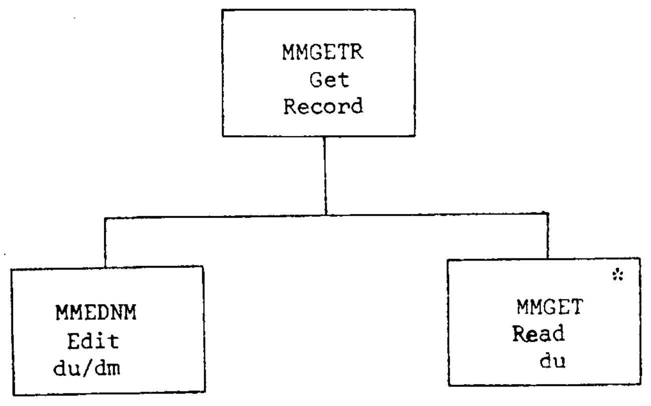
```
AND,     GET,     IFETCH,    MASK,
ØR,      REMARK,  SHIFT
```

Figure 7. MMGET Hierarchy Chart



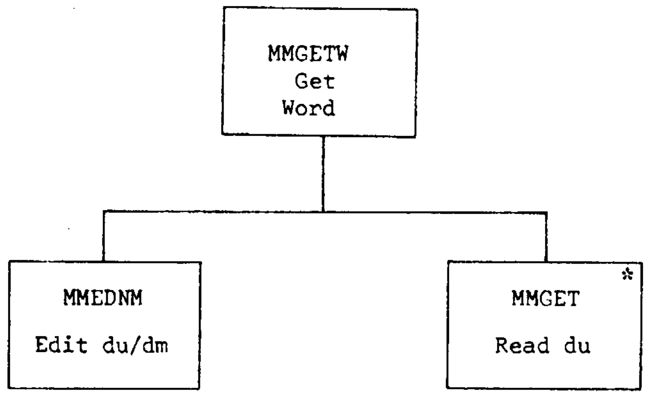
ANOPP Modules Called:
ILØC, MMERR, NWDTYP,
XFAN

Figure 8. MMGETE Hierarchy Chart



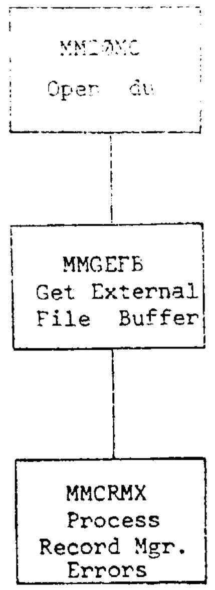
ANOPP Modules Called:
ILØC, MMERR, XFAN

Figure 9. MMGETR Hierarchy Chart



ANOPP Modules Called:
ILØC, MMERR, XFAN

Figure 10. MMGETW Hierarchy Chart



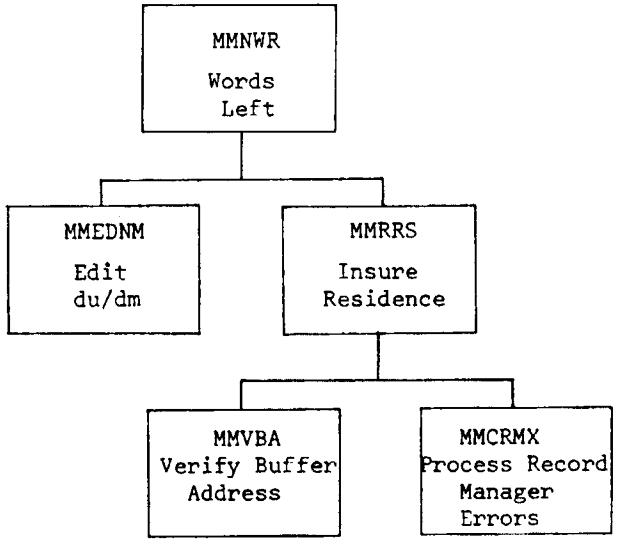
ANOPP Modules Called:

DSMG, ILØC, MMERR

CDC System Library Subprograms Called:

FILEWA, IFETCH, ØPENM, REMARK

Figure 11. MMIØMC Hierarchy Chart



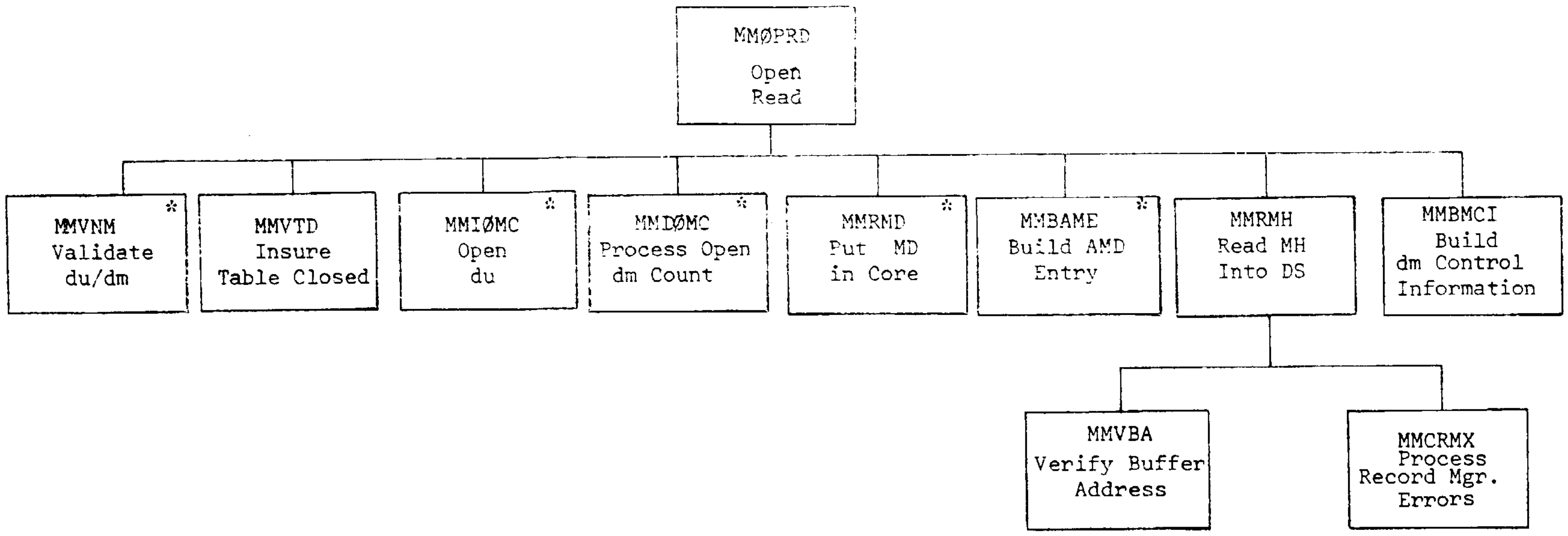
ANOPP Modules Called:

ILØC, MMERR, XFAN

CDC System Library Subprograms Called:

AND, GET, IFETCH, MASK, ØR,
REMARK, SHIFT,

Figure 12. MMNWR Hierarchy Chart



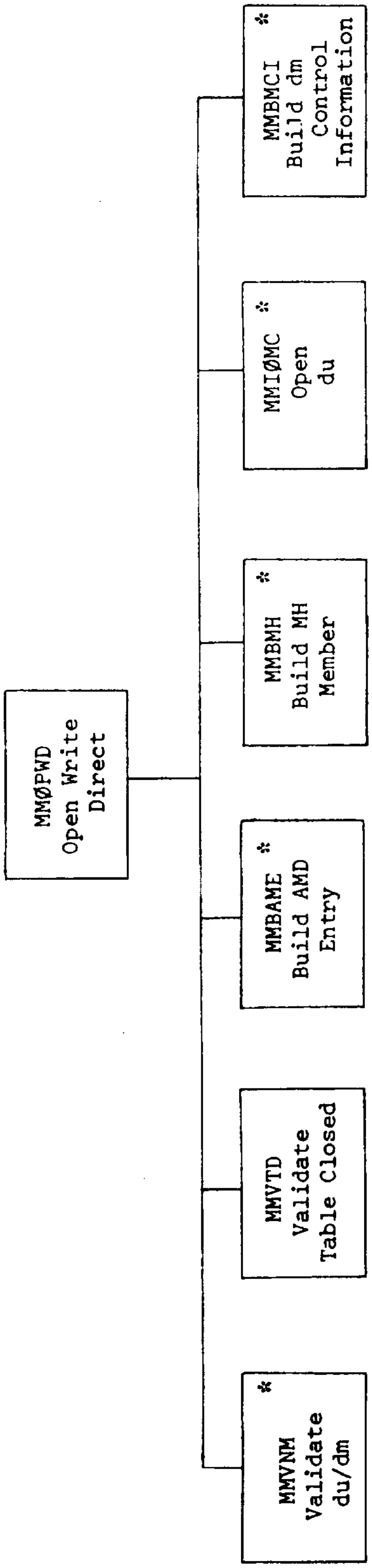
ANOPP Modules Called:

DSMG, DSMX, ILØC, MMERR, TMSTD, TMFTE, XT1FV

CDC System Library Subprograms Called:

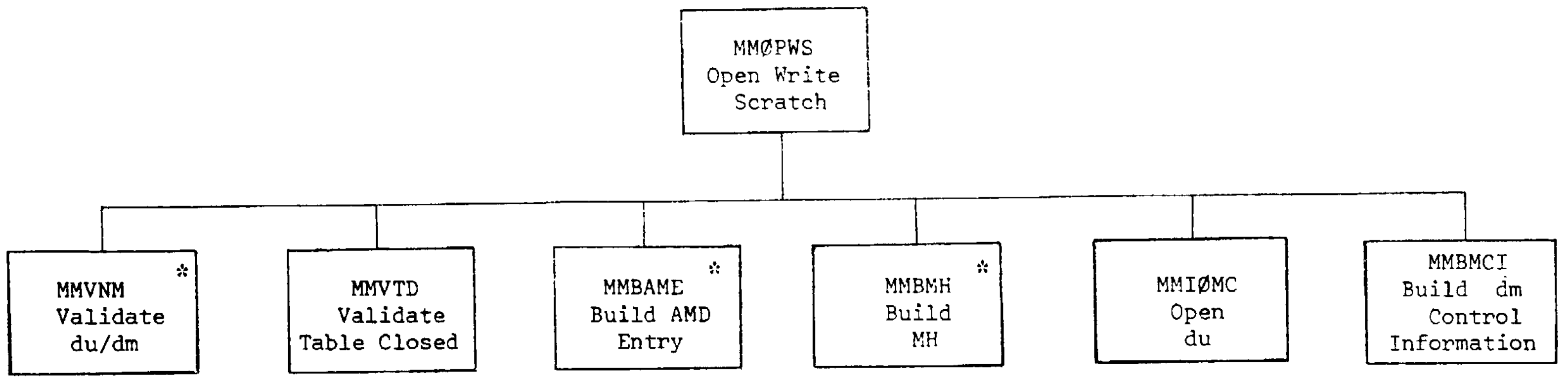
AND, FILEWA, GET, IFETCH, MASK, ØR, REMARK, SHIFT

Figure 13. MMØPRD Hierarchy Chart



ANOPP Modules Called:
MMERR, TMSTD, TMFTE

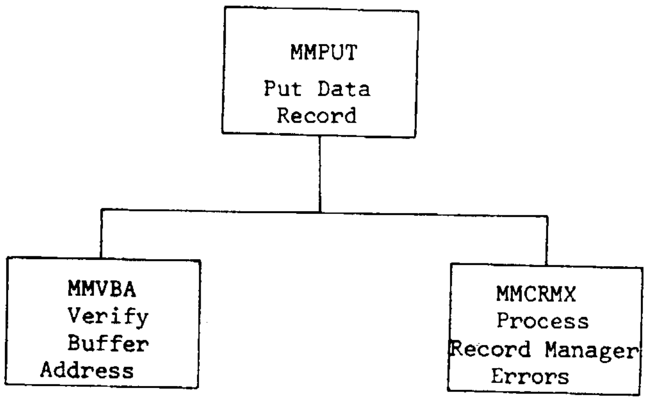
Figure 14. MMØPWD Hierarchy Charts



ANOPP Modules Called:

IØR, IRSHFT, MMERR, TMSTD, TMFTE, XCTDU, XCTEFN

Figure 15. MMØPWS Hierarchy Chart

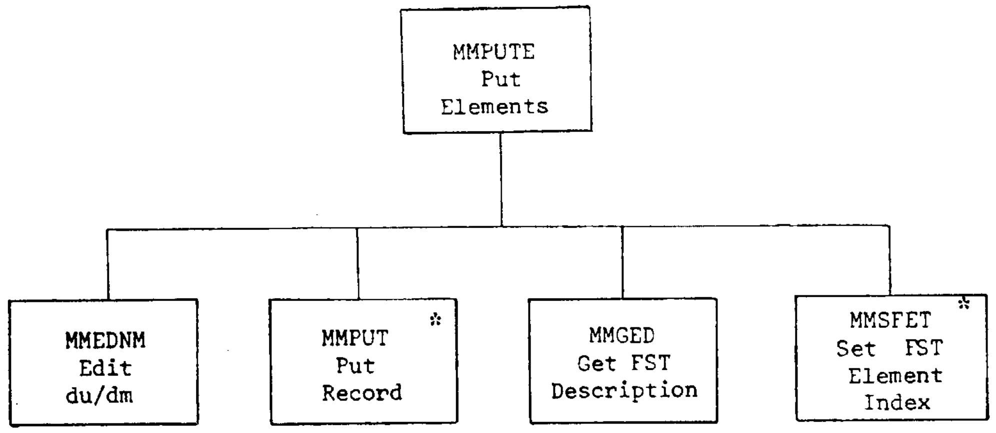


ANOPP Modules Called:
ILØC, MMERR

CDC System Library Subprograms Called:

AND,	IFETCH,	MASK,
ØR,	PUT,	REMARK,
SHIFT		

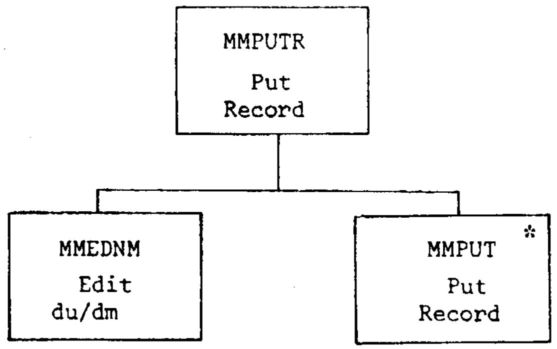
Figure 16. MMPUT Hierarchy Chart



ANOPP Modules Called:

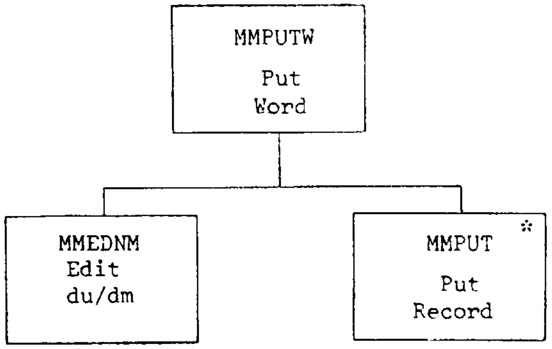
ILØC, MMERR, MWDTYP, XFAN

Figure 17. MMPUTE Hierarchy Chart



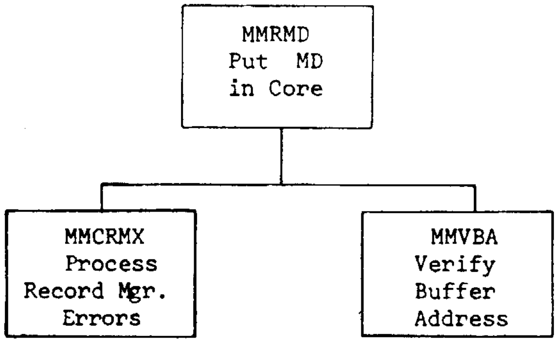
ANOPP Modules Called:
ILØC, MMERR, XFAN

Figure 18. MMPUTR Hierarchy Chart



ANOPP Modules Called:
ILØC, MMERR, XFAN

Figure 19. MMPUTW Hierarchy Chart



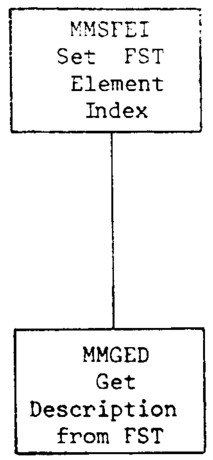
ANOPP Modules Called:

DSMX, ILØC, MMERR

CDC System Library Subprograms Called:

GET, AND, IFETCH, MASK,
ØR, REMARK, SHIFT

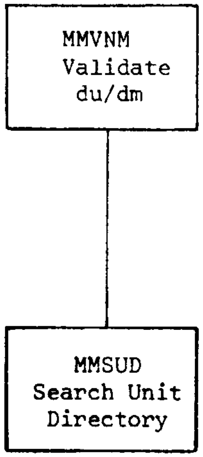
Figure 20. MMRMD Hierarchy Chart



ANOPP Modules Called:

MMERR, NWDTYP

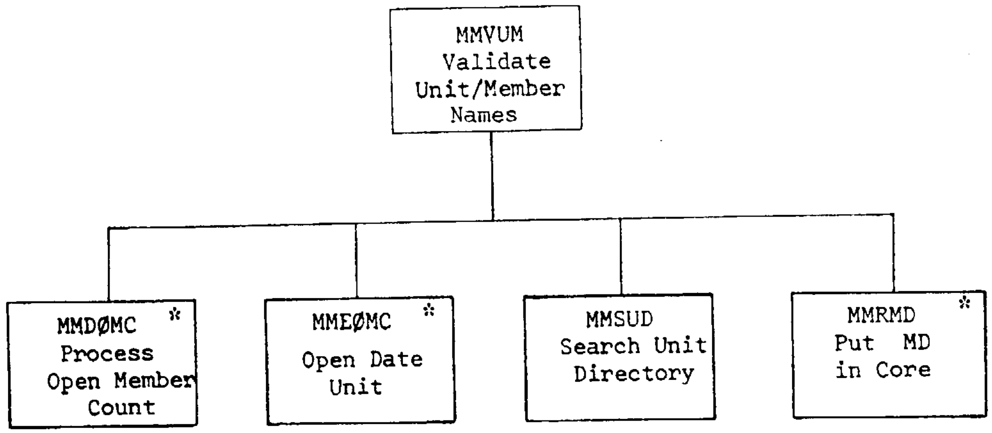
Figure 21. MMSFEI Hierarchy Chart



ANOPP Modules Called:

MMERR, XFAN, XVNAME, XT3FV

Figure 22. MMVNM Hierarchy Chart



ANOPP Modules Called:

MMERR, XFAN, XT1FV, XT3FV, XVNAME

Figure 23. MMVUM Hierarchy Chart

3.6.4 Data Table Manager

3.6.4.1 Overview

Data Tables are a special class of one-record members. To member manager they are unformatted, one-record members of a data unit. To table manager they are internally formatted table structures to be maintained in core while they are open. Allowance for several types of table structures will be made. The same member/table cannot be open simultaneously for processing by both table manager and member manager. The primary purpose of table manager is to maintain the table/member in core across the execution of several ANOPP modules and several openings and closings of the table.

The following is a general synopsis of events in the life of a table. When first opened, a table is read into Global Dynamic core and its name is entered into a table directory. When closed, the table remains in core and is logically closed in the directory. Subsequent opens will take place via the directory. If a table is altered during the time it is open in core, it should have been opened with an open alter so that a copy will be placed on the original member. This is necessary to preserve the integrity of the table under the following conditions: a) while a table is logically closed in the table directory, it can be removed from the directory either to make room for other tables or because member manager is processing the member for writing; b) when a table is removed from the directory, a subsequent open will read a new copy of the member into core and place its name in the table directory.

The following open, close, and interpolation routines work for any type table. Specific table build routines are supplied for specific table types. For further information about the structure of specific Data Table types, see Section 3.4.2.

All table manager routines require the NAME parameter to identify the data unit and member on which the table resides or will reside. NAME is a three word array with the following structure:

NAME(1)	unit name of the table (A8)
NAME(2)	member name of the table (A8)

NAME(3)	reserved word for use by table manager and other data management routines and not to be altered by user

3.6.4.2 Open Data Table Subroutines

Two open data table subroutines have been provided to give the user the ability to specify at open time whether a table is to be altered or not altered during processing. A data table that is opened with alter permission will be rewritten on its data member when closed. One open without alter permission is assumed to be intact and is not rewritten. When a data table is closed it is retained in global dynamic core as an inactive table so that subsequent opens need not reread it.

3.6.4.2.1 TMØPNA - Open with Alter Permission

Purpose: TMØPNA requsts that an ANOPP data table be opened with permission to alter.

Format: CALL TMØPNA (NAME)

Arguments:

NAME - a three word array containing the names of the data unit and member on which the data table resides. On return from TMØPNA, the third word contains the IDX to the data table in Global Dynamic core.

Description: TMØPNA calls subprogram TMTØPN passing the NAME argument and a logical alter flag which is set to true. TMTØPN performs validations to insure that the data table is not already open to Data Table Manager (DTM) or Data Member Manager (DMM). It then searches the inactive table chain in the Data Table Directory for the named table. If the table is found, its entry is linked into the active table chain. Otherwise, subprogram TMMØPN is called to read the table into global dynamic core and build an active table entry. The IDX of the table is then swapped into the third word of the NAME argument; the negation of the IDX of the NAME argument is stored in the DTD entry; and the table is opened for use.

Error Conditions: TMØPNA aborts with a message if the table is already open to Table Manager or Member Manager.

3.6.4.2.2 TMØPN - Open Without Alter Permission

Purpose: TMØPN requests that an ANOPP data table be opened without permission to alter.

Format: CALL TMØPN (NAME)

Arguments:

NAME - a three word array containing the names of the data unit and member on which the data table resides. On return from TMØPN, the third word contains the IDX to the data table in global dynamic core.

Description: TMØPN calls subprogram TMTØPN passing the NAME argument and a lcgical alter flag which is set to false. TMTØPN performs validations to insure that the data table is not already open to Data Table Manager (DTM) or Data Member Manager (DMM). It then searches the inactive table chain in the Data Table Directory for the named table. If the table is found, its entry is linked into the active table chain. Otherwise, subprogram TMMØPN is called to read the table in global dynamic core and build an active table entry. The IDX of the table is then swapped into the third word of the NAME argument; the IDX of the NAME argument is stored in the DTD entry; and the table is opened for use.

Error Conditions: TMØPN aborts with a message if the table is already open to Table Manager or Member Manager.

3.6.4.3 TMCLØS - Close Data Table Subroutine

Purpose: TMCLØS closes a data table.

Format: CALL TMCLØS (NAME)

Arguments:

NAME - a three word array containing the names of the data unit and member on which the data table resides, and the IDX to the data table.

Description: TMCLØS locates the entry for the data table in the active table chain in the Data Table Directory (DTD) and checks the NAME argument to insure that it matches the NAME argument used in opening the table. If the table was opened to alter, it is rewritten to its data member. It is then logically closed by swapping its IDX from the NAME argument into its DTD entry and linking the DTD entry into the inactive table chain.

Error Conditions: TMCLØS aborts with an error message if the table being closed was not open, if the closing name argument is not the same as the opening argument, or if the data structure being closed is not a data table.

3.6.4.4 TMTERP - Data Table Interpolation

Purpose: Retrieve an interpolated value of a dependent variable from a data table which is currently open.

Format: CALL TMTERP (NAME, ITYPE, X, Y, Z, ANS, ISTAT)

Arguments:

NAME - three word array with the following structure:
NAME(1) - unit name of the table
NAME(2) - member name of the table
NAME(3) - reserved; not to be altered by user.

ITYPE - type of interpolation required:
ITYPE = 0 no interpolation permitted
ITYPE = 1 linear interpolation requested

X,Y,Z values of the independent variables for which the corresponding dependent variable value is desired.

ANS - retrieved value if ISTAT = 1; otherwise contents unaltered.

ISTAT - status return:
ISTAT = 1 request complete; ANS contains dependent variable value desired
ISTAT = 0 request not completed; ANS does not contain dependent variable value desired.

Description: TMTERP module attempts to retrieve from the Data Table specified by NAME the desired dependent variable value.

TMTERP validates that (1) the table is open, (2) the table type found in the table is valid, (3) the interpolation procedure requested (ITYPE) is valid for this table, and (4) the number of independent variables found in the table is within range. If no error has been detected in the validations, an attempt is made to retrieve the desired value.

If the dependent variable is not in the table, an interpolated value will be returned if ITYPE does not equal zero. If the dependent variable is outside the range of the table, extrapolation procedures will or will not be used according to the extrapolation instructions found within the table. (These instructions are established by the user at the time the table is built.)

The user will be informed via ISTAT if the dependent variable value desired has been retrieved. The user must insure that the type of X, Y, Z, and ANS variable correspond to the variable types expected by the table.

Error Conditions: TMTERP prints an error message and returns to the caller with request not filled if the requested interpolation procedure is invalid. TMTERP aborts with a message if the number of independent variables found in the data table is out of range, if the table type is invalid, or if an error was detected in the table edit.

3.6.4.5 Data Table Building

Since data tables may be built in a functional module, subroutines are provided which permit the user to build tables of the type supported by table manager (see Section 3.4.2 Data Table Types). Tables may also be built in the control statement stream by using the TABLE CS (see the TABLE CS description in Section 3.5).

3.6.4.5.1 TMBLD1 - Build Type 1 Table

Purpose: Build data table of structure type 1.

Format: CALL TMBLD1 (NAME, NINT, INT, ITYPDV, NIND, IDSCRP, I1, I2, I3, IDV, IERR)

Arguments:

NAME - three word array with the following structure:

NAME(1) - unit name of table
NAME(2) - member name of table
NAME(3) - reserved; not to be altered by user.

NINT - number of elements in array containing interpolation procedures acceptable on this table.

INT - array containing integer codes of interpolation procedures acceptable on this table. Valid codes are:

0 - no interpolation
1 - linear interpolation

ITYPDV - type of dependent variable. Valid values are:
1 - integer
2 - real single precision
3 - real double precision

NIND - number of independent variables in this table -- 1, 2 or 3.

IDSCRP - array of dimension NIND*4 containing a description of the NIND independent variables.

(4*I-3) FØRMAT of the Ith independent variable:
0 - ordered position from one to IDSCRP(4*I-2). This variable value array does not exist
1 - integer
2 - real single precision
3 - real double precision

(4*I-2) Integer number of Ith independent variable (.GE.0)

(4*I-1) Interpolation procedure if value desired is greater than the largest value of the Ith independent variable
0 - no interpolation
1 - use closest independent variable value
2 - extrapolate (linear)

(4*I) Interpolation procedures if value desired is less than the smallest value of the Ith independent variable (same values as above)

I1,I2,I3 - start location of arrays containing independent variable values as defined for data table type 1. If an independent variable is format type zero for ordered position, then the corresponding array is ignored.

IDV - NIND dimensional array of dependent variable values.

IERR - indicates if table was built:
IERR = 0 no error detected; table was built
IERR = -1 error detected; table was not built

Description: The TMBLD1 module builds a Data Table Type 1. The input values NINT, INT, ITYPDV, NIND, and IDSCRP are edited to insure they are in the range expected. The independent variable arrays are validated to insure they are in monotonic sequence. If no errors are detected in the edit, a Type 1 table structure is generated and put on the unit/member specified by NAME. The table is not maintained in core but is available for processing via TMØPN. If a duplicate table exists, it is replaced. A table by this name may not currently be open either by a table manager or a member manager call. The user is informed via IERR if an error occurred which prevented the building of the table.

Error Conditions: TMBLD1 writes an error message if the unit the table was to be built on is not in the unit directory, if there is insufficient core to build a table, or if errors were detected in editing the values to be used in building the table.

3.6.4.6 Auxiliary Modules

An auxiliary module performs a function common to all table manager modules and is available for use by these modules.

3.6.4.6.1 TMERR - Table Manager Error Message Writer

Subroutine TMERR (NUM, NAME, IVAL, IRAY, L) processes fatal and non-fatal errors for the table manager modules and the DBM control statement TABLE (XTB). NUM, the integer number of the error message to be printed, is negative if the error is fatal and positive if the error is non-fatal. TMERR prints the informative error message (indicated by the absolute value of NUM) with the specific value(s) causing the error condition (indicated by input values NAME, IVAL, IRAY). If the error is fatal, ANOPP is aborted by a call to XEXIT. If the error is non-fatal, a traceback is performed to the major table manager module called by the user.

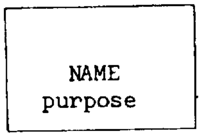
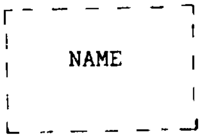
3.6.4.7 Hierarchy Charts

A hierarchy chart is a graphical representation of the logical relationships between modules. Figures 24-27 are the hierarchy charts for the table manager modules and the auxiliary module.

In general, only table manager modules appear as a block entity in the charts and all table manager modules appear at least once. The charts are in alphabetical order with respect to module name except for Figure 24 which represents the logical grouping of the table manager modules. A hierarchy for the auxiliary module is also among the alphabetized charts.

A module which is not part of table manager but is called by a table manager module is not shown as a block entity, but is listed at the bottom of the chart. The module may be an ANOPP executive module which is part of Member Manager, the Dynamic Storage Management System, or the General Utilities. It may also be a subprogram provided by one of the CDC operating system libraries. In either case, the module is generally of a service or utility nature and may be called many times by various table manager modules.

Symbols and headings used in the hierarchy charts are given below:

	NAME - module name purpose - brief description
	represents logical module not existing as entity. It is used for logical groupings.
————————	indicates lower module is called by the higher module.
----------	implies logical grouping with no direct relationship.
*	In upper right corner of module block indicates module is expanded as a separate entity

ANOPP Modules Called: a list of DBM, DSM, and General Utility modules called by the modules in this figure.

CDC System Library Subprograms Called: a list of subprograms called by the modules in this figure and which are not part of ANOPP but are provided by CDC NOS operating system libraries.

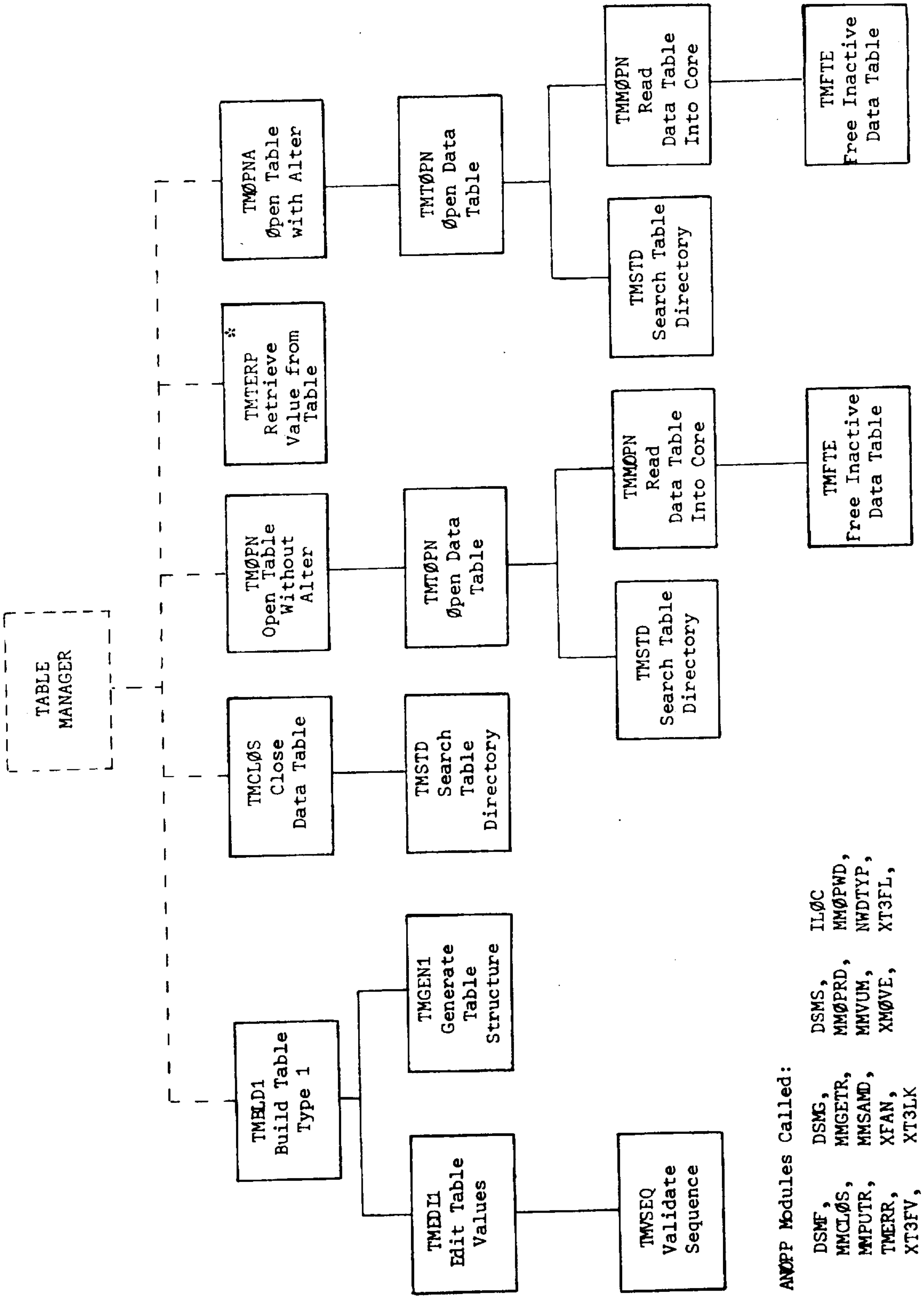


TABLE MANAGER
TMBLD1 Build Table Type 1
TMGEN1 Generate Table Structure
TMEDI1 Edit Table Values
TMVSEQ Validate Sequence
TMCLØS Close Data Table
TMSTD Search Table Directory
TMØPN Open Table Without Alter
TMTØPN Øpen Data Table
TMSTD Search Table Directory
TMMØPN Read Data Table Into Core
TMFTE Free Inactive Data Table
TMTERP * Retrieve Value from Table
TMØPNA Øpen Table with Alter
TMTØPN Øpen Data Table
TMSTD Search Table Directory
TMMØPN Read Data Table Into Core
TMFTE Free Inactive Data Table

ANØPP Modules Called:

DSMF, DSMG, DSMS, ILØC
MMCLØS, MMGETR, MMØPRD, MMØPWD,
MMPUTR, MMSAMD, MMVUM, NWDTYP,
TMERR, XFAN, XMØVE, XT3FL,
XT3FV, XT3LK

Figure 24. Table Manager Hierarchy Chart

TMERR
Error
Message
Writer

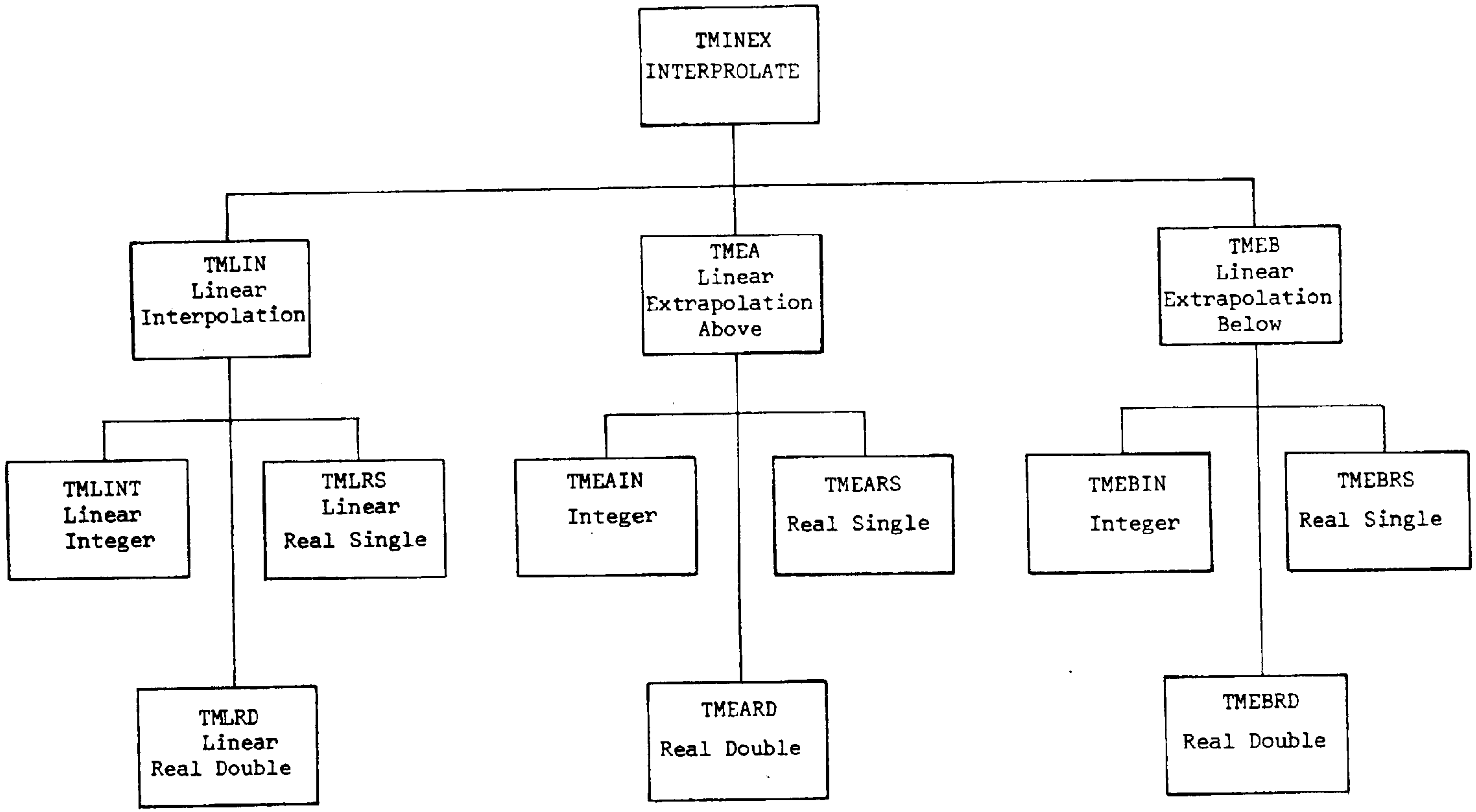
ANOPP Modules Called:

XEXIT, XFETCH, XPLINE,
XTBDMP, XTRACE

CDC System Library Subprograms Called:

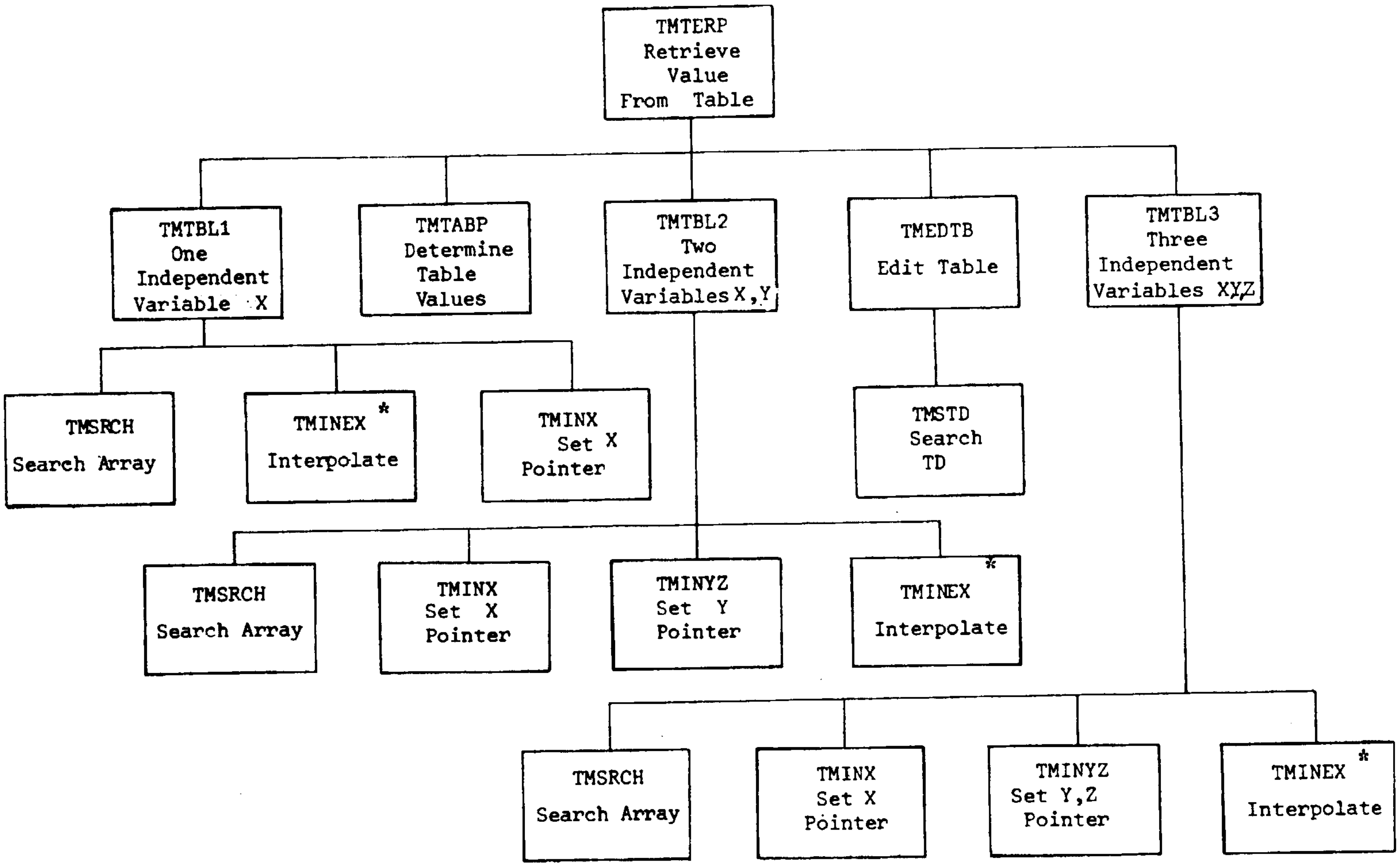
IABS

Figure 25. TMERR Hierarchy Chart



ANOPP Subprograms Called:
TMERR, XINC

Figure 26. TMINEX Hierarchy Chart



ANOPP Modules Called:

ILØC, NWDTYP, TMERR, XBSRIN,
XBSRRD, XBSRRS, XFAN, XT3FV

CDC Library System Subprograms Called:

IABS

Figure 27. TMTERP Haerarchy Chart

3.7 DYNAMIC STORAGE MANAGEMENT SYSTEM (DSM)

3.7.1 Overview

The ANOPP Dynamic Storage Manager (DSM) provides a method of allocating and releasing core storage within ANOPP. The boundaries of the storage area to be managed are defined to be from the end of the longest overlay segment of the presently executing executive or functional module to the last word of the field length.

Obviously, as different executive and functional modules are called into execution, the available free storage fluctuates. Therefore, in order to provide for executive inter-module communication and for the storage of ANOPP directories and tables, a section of free storage, known as "Global Dynamic Storage" (GDS), has been defined. The starting address of GDS is established by the Executive Initialization Phase (XBS) and must be beyond the longest segment of the largest module which will be executed by a particular ANOPP run.

The rest of the free storage is available for intra-module usage as "Local Dynamic Storage" (LDS). LDS begins with the word following the longest segment of a particular module and ends at the start of GDS.

Addressing of Dynamic Storage by the user is accomplished by indexing relative to a fixed common block - XANØPP. The FORTRAN statement CØMMØN/XANØPP/IX(1) must be included in every program and subroutine that directly references dynamic storage. DSM will then return an index (IDX), relative to XANØPP, whenever a block of dynamic storage is reserved (DSMG). The variable containing the index is thereafter reserved for DSM use. The contents of IDX must not be changed by the user unless the reserved block has been released (DSMF) or another variable is provided to DSM (DSMS).

At the beginning of execution, all core in the dynamic storage areas belong to two free blocks, one for LDS and one for GDS. As blocks are reserved and released during execution, dynamic storage will be divided into a number of separate blocks, some reserved and some free. To minimize fragmenting of dynamic storage, DSM will collapse a block of dynamic storage into adjacent free blocks, when it is released, to form a single free block.

Still, fragmentation of free blocks may reach the point where a single block of the requested size cannot be reserved for the user. When this occurs, the free blocks will be consolidated into one free block. This occurs automatically without the user's knowledge and will involve the relocation of reserved blocks and the updating of their respective indices.

If the user wishes to inhibit consolidation, he may "lock" dynamic storage using a DSM utility (DSML). When the situation requiring the lock is passed, the user must then "unlock" dynamic storage (DSMU) to re-enable consolidation. The results of absolute addressing schemes in unlocked dynamic storage are unpredictable and best avoided.

Finally, DSM provides a utility to expand a reserved block (DSMX). DSMX will attempt to expand a reserved block by reserving any adjacent free storage. If that is not possible, DSMX will reserve a new, larger block, relocate the contents of the original block, and update the user's IDX. DSMX will relocate the specified block regardless of whether dynamic storage is locked or unlocked.

3.7.2 Dynamic Storage Structure

The Dynamic Storage Management System maintains two distinct storage areas in the free storage area defined as the block of core from the end of the longest overlay segment of the presently executing executive or functional module to the last word of field length.

The area known as Global Dynamic Storage (GDS) begins somewhere beyond the longest segment of the largest module executed during the ANOPP run, and ends with the last word of field length. The length of GDS is determined by the LENGL parameter on the ANØPP control statement, if specified, or a system default that provides a minimum length necessary for ANOPP to run. GDS is established during the Initialization Phase (XBS) for the life of the ANOPP run.

The area known as Local Dynamic Storage (LDS) begins with the first word following the overlay segment in current execution and uses all storage not already allocated for GDS. LDS is initialized by each functional or executive module according to the length of its overlay segment. The positioning of LDSA is accomplished by loading a labeled common block after the longest overlay segment of the module. The definition and placement of this common block is the responsibility of the user. The name of this common block must be unique and by convention begins with LDSA. Each functional or executive module that initializes LDS must also release LDS before control passes to another module.

The structure of the two storage areas LDS and GDS is identical. Each has three header control words and a single trailer control word that identify the storage type (GDS or LDS), give the pointer to the first block in the free storage chain, and delimit the storage area (see Dynamic Storage Control Words Section).

As the free storage in GDS and LDS is reserved by users, and subsequently released, the free storage becomes fragmented. In order to keep track of these fragmented free storage blocks, each storage area maintains its own free storage chain. As part of this chain, each free block contains a forward and a backward pointer to other blocks in the chain.

3.7.2.1 Dynamic Storage Control Words

The Dynamic Storage control words include three header words and a single trailer word on both Global Dynamic Storage and Local Dynamic Storage. These header and trailer control words are initialized by the DSMI module at the time the Global Dynamic and Local Dynamic storage areas are initialized by DSMI.

The header control words are the three words beginning with the first word of the dynamic storage area (GDS or LDS). Word 1 is initialized by DSMI with the dynamic storage type (3HGDS or 3HLDS). Word 2 is initialized by DSMI with the relative address of the first block in the free storage chain. Word 3 is initialized by DSMI with a convenient bit pattern (all bits on) for delimiting the storage area.

The trailer control word is also initialized by DSMI with a convenient bit pattern (all bits on) for delimiting the storage area.

EXAMPLE:

IX(SADDR)	LDS or GDS
IX(SADDR+1)	SADDR+3
IX(SADDR+2)	77777777777777777777777777777777777777
Storage words available to user	(This area holds blocks reserved by users of the system, plus the free storage chain.)
IX(EADDR)	77777777777777777777777777777777777777

where SADDR and EADDR are parameters from the /XDSMC/ common block.

SADDR is the relative address of the start of dynamic storage area.

EADDR is the relative address of the end of dynamic storage area.

3.7.2.2 Reserved Block Control Words

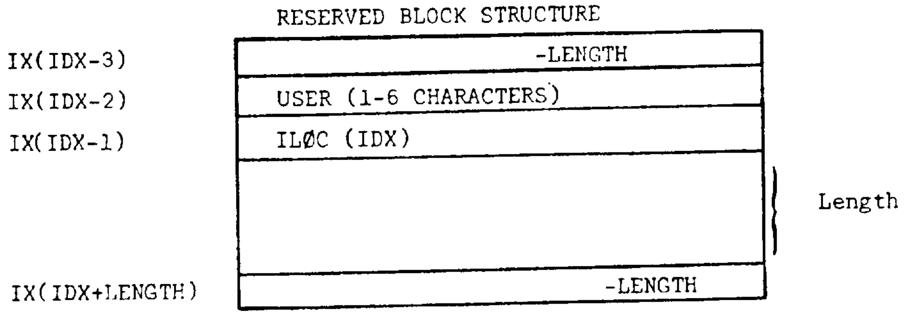
The reserved block control words include three header words at the beginning of each reserved block and one trailer word at the end of the block.

Word 1 of the header control words of a reserved block is defined as the complement of the length of the reserved block. In this case, length does not include the reserved block control words. Word 2 of the header control words is defined as the name (1-6 characters) of the user that reserved the block. User name is taken from the USER parameter on the DSMG call to reserve the block Word 3 of the header control words is set to the relative address of the user's IDX variable.

The trailer control word of a reserved block, like the first header control word, is set to the complement of the block length.

Reserved block control words are defined by the DSMG module when the block is reserved.

EXAMPLE:



where ILOC is an integer function that determines the address of the IDX variable relative to /XANØPP/.

Blocks may be reserved in either Global Dynamic Storage or Local Dynamic Storage. After initialization of Global Dynamic Storage and Local Dynamic Storage, and one block has been reserved with IDX = IDXA and length LENGTHA in GDS, the storage area appears as follows:

IX(LSADDR)	LDS	
IX(LSADDR+1)	LSADDR+3	
IX(LSADDR+2)	777777777777777777777777777777777777777	
	FREE STORAGE	
IX(LEADDR)	777777777777777777777777777777777777777	
IX(GSADDR)	GDS	
IX(GSADDR+1)	GSADDR+3+LENGTHA+4	
IX(GSADDR+2)	777777777777777777777777777777777777777	
IX(IDXA-3)	-LENGTHA	
IX(IDXA-2)	USER	
IX(IDXA-1)	ILØC(IDXA)	
	RESERVED BLOCK	LENGTHA
IX(IDXA+LENGTHA)	-LENGTHA	
	FREE STORAGE	
IX(GEADDR)	777777777777777777777777777777777777777	

where ILØC returns the address of the IDXA variable relative to /XANØPP/ and where /XDSMC/ common block parameters have the following definitions:

LSADDR - LDS start relative to /XANØPP/
LEADDR - LDS end relative to /XANØPP/
GSADDR - GDS start relative to /XANØPP/
GEADDR - GDS end relative to /XANØPP/.

3.7.2.3 Free Storage Control Words

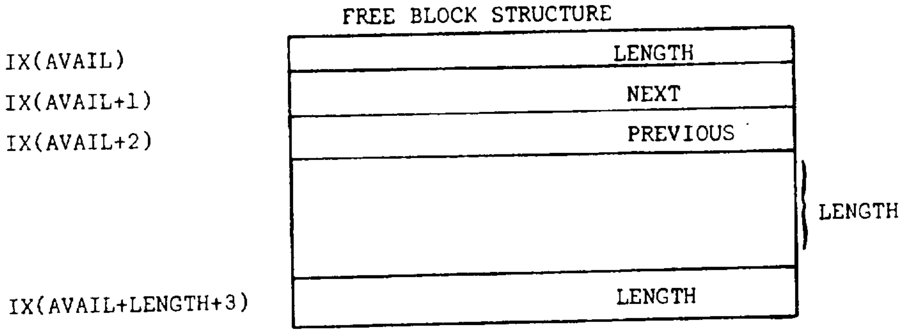
Free storage in GDS and LDS is maintained in a chain for internal control. Therefore, each free block header contains a forward pointer and a backward pointer that establishes the block's place in the chain. The free block control words include a three-word header and a trailer word.

Word 1 of the header control words contains the length of the free storage block. This length excludes the free block control words. Word 2 of the header contains the address relative to /XANØPP/ of the next block (forward pointer) in the free chain. Word 3 of the header contains the address relative to /XANØPP/ of the previous block in the chain (backward pointer).

The trailer control word of a free block, like word 1 of the header, contains the length of the block. This length excludes control words.

When either storage area GDS or LDS is initialized, all words in that storage area belong to the free storage chain, which has only one block. The DSMI module defines the control words for the single block in the free storage chain as well as the control words for the storage area.

EXAMPLE:



where AVAIL is the address relative to /XANØPP/ of free block

3.7.3 Dynamic Storage Management System User Modules

Although dynamic storage is intended for use by the ANOPP Executive and Data Management Systems, functional modules may also make use of DSM. The calling sequence for most DSM subroutines is as follows:

CALL DSMx (USER, TYPE, IDX, $P_1 \ldots, P_n$)

USER - integer variable defining name of calling module as a one to six character Hollerith constant

TYPE - defines the dynamic storage type (LDS or Local Dynamic Storage and GDS for Global Dynamic Storage)

IDX - user defined integer variable which will contain the location of a block of dynamic storage relative to a reference point. For the ANOPP system, the reference point is the /XANØPP/ common block. The address of IDX will be stored in the block of dynamic storage reserved for USER, and the index stored at that address will be updated whenever the dynamic storage block is moved due to a consolidation request.

P_1-P_n - miscellaneous parameters required by the specific DSM module.

The USER parameter required by most DSM modules is the key to ownership of a reserved block of storage. When a storage block is reserved, the name of the user making the request is stored in the block. Subsequent operations on that block of storage are permitted only for the appropriate user.

3.7.3.1 DSMB - Determine Dynamic Storage Boundaries

Purpose: To retrieve the start and end addresses of Local Dynamic Storage (LDS) for subsequent initialization of LDS by DSMI.

Format: CALL DSMB(A)

Arguments:

A - array in labeled common LDSAxxx, supplied by the user, which has been reserved for Local Dynamic Storage. On output, A(1) contains the index relative to /XANØPP/ to start of Local Dynamic Storage. A(2) contains the index relative to /XANØPP/ to end of Local Dynamic Storage.

Description: The DSMB module should be called when a module is integrated into the ANOPP system to get the start and end addresses of Local Dynamic Storage. On entry, Global Dynamic Storage must already be initialized.

Labeled common LDSAxxx is loaded immediately after the user's longest module to give him full benefit of the available storage. DSMB calculates the start of the array A in labeled common LDSAxxx as an index relative to /XANØPP/ and stores the index in the first word of the array. This is the index to the beginning of LDS.

DSMB calculates the end of LDS as an index relative to /XANØPP/ and stores the index in A(2). The end of LDS is calculated as the address of the word immediately preceeding the start of Global Dynamic Storage.

With the start and end address of LDS stored in the first and second word of array A, the user may initialize LDS via DSMI.

Error Conditions: DSMB aborts with a message if LDS and GDS boundaries overlap.

3.7.3.2 DSMD - Dynamic Storage Dump

Purpose: To dump the contents of Global or Local Dynamic Storage or of a single reserved block.

Format: CALL DSMD (USER, TYPE, IDX)

Arguments:

USER - one to six-character name of user for dynamic storage area or reserved block.

TYPE - three-character code indicating storage type (3HGDS or 3HLDS).

IDX - index relative to IX of reserved block to be dumped, or zero, if entire storage area is to be dumped.

Description: The DSMD module should be called to dump all dynamic storage or a single reserved block. Prior to performing the dump, DSMD validates the USER and TYPE arguments for a dynamic storage dump or USER, TYPE, and IDX for a reserved block dump.

DSMD prints the contents of the dynamic storage area control words including storage type, first free block pointer, dynamic storage, start address, and end address. Then the contents of the storage area or the reserved block are dumped. For the dump of an entire storage area, only the contents of control words are printed for free blocks, while all words in reserved blocks are printed.

Error Conditions: DSMD aborts with a message if the dynamic storage type is not initialized, if the dynamic storage type is invalid, if the user is invalid, or if the IDX is invalid.

3.7.3.3 DSMF - Free a Reserved Block of Dynamic Storage

Purpose: To free a block of dynamic storage previously reserved by a call to DSMG.

Format: CALL DSMF(USER, TYPE, IDX)

Arguments:

USER - one to six-character name of user requesting to free the block residing at the address specified by IDX in storage area identified by TYPE.

TYPE - three-character code identifying storage area where block is to be freed resides.

IDX - index relative to /XANØPP/ for reserved block that is being freed.

Description: The DSMF module frees a reserved block and returns it to the free storage chain making it available for use. However, before freeing the block, DSMF validates that the user freeing the block is the same user that reserved the block and that the storage type and IDX are valid.

The header and trailer control words of the reserved block (see Section 3.7.2) are defined to make the block part of the free storage chain. The control words will reflect block size and forward and backward pointers to the free storage chain.

If either, or both, of the blocks bordering the newly freed block is(are) also a free block, the newly freed block is collapsed into the adjacent free block to form a larger free block. Control words for the newly formed block are redefined to reflect the new size and adjustment in the forward and backward pointers to the chain.

Error Conditions: DSMF aborts with a message if there is an invalid user, dynamic storage type or IDX.

3.7.3.4 DSMG - Get A Block of Dynamic Storage

Purpose: To obtain a block of dynamic storage from the free storage chain, reserve it for the user, and return the IDX of the reserved block.

Format: CALL DSMG(USER, TYPE, IDX, MIN, MAX, LEN)

Arguments:

USER - one to six-character name of user reserving a block.

TYPE - three-character code indicating storage area where block should be reserved (3HGDS or 3HLDS).

IDX - (OUTPUT) address relative to /XANØPP/ for reserved block. IDX points to first usable word of block (first word following reserved block header).

MIN - minimum number words of dynamic storage required by user.

MAX - maximum number words of dynamic storage desired by user.

LEN - (OUTPUT) length of dynamic storage block reserved for user. Length excludes reserved block control words. If there was an insufficient amount of free storage available to satisfy user's request, the LEN parameter is set to zero on output.

Description: The DSMG module attempts to locate and reserve a block of dynamic storage that will satisfy the user's requirements. DSMG searches the free storage chain for a free block that will satisfy the user's maximum request.

If such a block is found, the maximum number of words requested is reserved for the user, and the reserved block control words are defined accordingly. The address relative to /XANØPP/ of the reserved block is returned to the user in the IDX variable.

If the entire free block is used in allocating the reserved block, then the free block is removed from the free storage chain. In order to remove a block from the free chain, the forward pointer of the preceeding block in the chain and the backward pointer of the next block in the chain are redefined to point to each other. This completely removes the block from the free chain.

EXAMPLE: Assume that on entry to DSMG Local Dynamic Storage has the following configuration.

IX(LSADDR)	LDS
	AVAILA(1ST FREE BLOCK)
	7777777777777777777777777777777777777
IX(AVAILA)	LENGTHA
	AVAILC(FORWARD POINTER)
	LSADDR(BACKWARD POINTER)[1]
	FREE BLOCK A OF LENGTH LENGTHA
	LENGTHA
	-LENGTHB
	USERB
	ILØC(IDXB)
IX(IDXB)	RESERVED BLOCK B OF LENGTH LENGTHB
	-LENGTHB
IX(AVAILC)	LENGTHC
	ZERO (FORWARD POINTER)[2]
	AVAILA (BACKWARD POINTER)
	FREE BLOCK C OF LENGTH LENGTHC
	LENGTHC
IX(LEADDR)	777777777777777777777777777777777777777

[1] The backward pointer of the first block in the free chain always points to the start of the dynamic storage area.

[2] The forward pointer of the last block in the free chain always contains zero.

Also assume that the user requests a block where MIN = MAX = LENGTHA. Then DSMG uses all of free block A and removes block A from the free chain. Then LDS has the following configuration. An * indicates a changed entry.

IX(LSADDR)	LDS	
	AVAILC	*
	77777777777777777777777777777777777777	
	-LENGTHA	*
	USERA	*
	ILØC(IDXA)	*
IX(IDXA)	RESERVED BLOCK A OF LENGTH LENGTHA	
	-LENGTHA	*
	-LENGTHB	
	USERB	
	ILØC(IDXB)	
IX(IDXB)	RESERVED BLOCK B OF LENGTH LENGTHB	
	-LENGTHB	
IX(AVAILC)	LENGTHC	
	ZERO (FORWARD POINTER)	
	LSADDR (BACKWARD POINTER)	*
	FREE BLOCK C OF LENGTH LENGTHC	
	LENGTHC	
IX(LEADDR)	77777777777777777777777777777777777777	

In the example above, ILØC is a function that returns an address relative to /XANØPP/, and /XDSMC/ parameters are defined as follows:

LSADDR - LDS start address relative to /XANØPP/

LEADDR - LDS end address relative to /XANØPP/

However, if only a portion of the free block is used in allocating a reserved block, then the free block maintains its place in the free chain but is reduced in size. Control words for the reduced free block are redefined to reflect the block's reduced size. In addition, the forward pointer of the preceeding block in the chain and the backward pointer of the next block in the chain are redefined to reflect the reduced free block's next location.

EXAMPLE: Assume that on entry to DSMG local dynamic storage looks exactly like it did at the beginning of the previous example. Also assume that the user requests a block where MIN = MAX $<$ LENGTHA. Then DSMG reserves only a portion of free block A, and LDS has the following configuration.

DYNAMIC STORAGE MANAGEMENT SYSTEM (DSM)

IX(LSADDR)	LDS	
	AVAILA'	*
	777777777777777777777777777777777777	
	-LENGTHX	*
	USERX	*
	ILOC(IDXX)	*
IX(IDXX)	RESERVED BLOCK X OF LENGTH LENGTHX	
	-LENGTHX	*
IX(AVAILA')	(LENGTHA-LENGTHX-4)	*
	AVAILC (FORWARD POINTER)	
	LSADDR (BACKWARD POINTER)	
	REDUCED FREE BLOCK A OF LENGTH LENGTHA-LENGTHX-4	*
	(LENGTHA-LENGTHX-4)	*
	-LENGTHB	
	USERB	
	ILOC(IDXB)	
IX(IDXB)	RESERVED BLOCK B OF LENGTH LENGTHB	
	-LENGTHB	
IX(AVAILC)	LENGTHC	
	ZERO (FORWARD POINTER)	
	AVAILA' (BACKWARD POINTER)	
	FREE BLOCK C OF LENGTH LENGTHC	
	LENGTHC	
IX(LEADDR)	777777777777777777777777777777777777	

where ILØC and /XDSMC/ parameters LEADDR and LSADDR are defined in the previous example.

If attempts to find a free block that will satisfy user's maximum request fail, but there are enough words in all fragmented free blocks combined to provide at least the minimum request, then a consolidation of free storage is performed.

After consolidation, the largest block in the free chain is examined to see if it satisfies the maximum request. (Note that if the dynamic storage area was locked, the consolidation is a null process and the storage area is unchanged. However, if the consolidation is actually accomplished, then all available words in the free chain have been consolidated into a single block.) If the free block satisfies the maximum request, all or part of the block is reserved for the user. If the free block satisfies only minimum, or some value between minimum and maximum, then all or part of the free block is reserved accordingly.

If all attempts fail to satisfy the user's request, the block length LEN returned to user is set to zero.

Error Conditions: DSMG aborts with a message if an invalid storage type is requested, if the storage type requested is not initialized, if the minimum block size requested exceeds maximum block size requested, if LDS or GDS has been overlayed, or if the minimum or maximum block size requested is negative or zero.

3.7.3.5 DSMI - Initialize Dynamic Storage

Purpose: To initialize the control words for the specified dynamic storage type. In addition, to initialize the control words for the single free block in the free storage chain for the dynamic storage type initialized.

Format: CALL DSMI(USER, TYPE, START, END)

Arguments:

USER - one to six-character name of user requesting initialization.

TYPE - three-character code indicating storage type to be initialized (3HGDS or 3HLDS).

START - integer variable containing the absolute address of the start of dynamic storage.

END - integer variable containing the absolute address of the end of dynamic storage.

Description: The DSMI module must be called to initialize a dynamic storage area before that storage area may be used. DSMI initializes the dynamic storage control words described in Section 3.7.2.1 Dynamic Storage Control Words, and the free block control words described in Section 3.7.2.3 Free Storage Control Words.

Error Conditions: DSMI aborts with a message if the storage area being initialized has already been initialized, if the start address for the storage area is invalid, if the user requesting initialization is invalid, if there is insufficient storage length for initialization, or if LDS/GDS boundaries overlap.

3.7.3.6 DSML - Lock Dynamic Storage

Purpose: To prohibit consolidation of fragmented dynamic free storage.

Format: CALL DSML(USER, TYPE)

Arguments:

USER - one to six-character name of user imposing lock condition on dynamic storage.

TYPE - three character code indicating dynamic storage on which lock condition is being imposed.

Description: The DSML module increments the user lock against consolidation of free storage for the dynamic storage area specified. However, before incrementing the user lock, DSML does a consolidation of free storage. The consolidation relocates all reserved blocks to contiguous words of storage immediately following the storage area's control words. This forces all free words of storage, however fragmented, into a single free block.

If DSML has been called previously, such that the user lock has already been incremented, then the consolidation becomes a null process and DSML increments the user lock again.

Error Conditions: DSML aborts with a message if the storage type is invalid.

3.7.3.7 DSMQ - Query to Obtain Size of Largest Available Block

Purpose: To return the length of the largest amount of free storage available.

Format: CALL DSMQ(USER, TYPE, LEN)

Arguments:

USER - one to six-character name of user making inquiry.

TYPE - three-character code indicating dynamic storage area being queried.

LEN - (OUTPUT) length of largest amount of free storage available in storage area specified. The LEN parameter will be set to zero if no free storage is available.

Description: The DSMQ module returns the amount of free storage available to the user. If the storage area specified is locked against consolidation, the length returned is the length of the largest free block available. If the storage area is not locked against consolidation, DSMQ will consolidate the free storage and return the length of the resulting free block.

Error Conditions: DSMQ aborts with a message if the storage type is invalid.

3.7.3.8 DSMR - Release Dynamic Storage

Purpose: To release dynamic storage previously initialized via DSMI.

Format: CALL DSMR(USER, TYPE)

Arguments:

USER - one to six-character name of user releasing dynamic storage.

TYPE - three-character code indicating dynamic storage type (3HGDS or 3HLDS).

Description: The DSMR module releases dynamic storage previously initialized via the DSMI module. The user must call DSMR to release Local Dynamic Storage before a new overlay segment is introduced and control passes to another user.

Users are not prevented from releasing Global Dynamic Storage, but unpredictable results are certain to occur since the ANOPP system uses GDS for its directories and tables.

Releasing a storage area causes the user lock against consolidation to be cleared.

Error Conditions: DSMR aborts with a message if the user does not own the area of storage being released or if the storage type is invalid.

3.7.3.9 DSMS - Swap IDX Variables

Purpose: To change the address of an IDX maintained by Dynamic Storage Management System to another address.

Format: CALL DSMS(USER, TYPE, ØIDX, NIDX)

Arguments:

USER - one to six-character name of user swapping IDX variables.

TYPE - three-character code indicating dynamic storage type (3HGDS or 3HLDS).

ØIDX - ("old IDX") the IDX variable that is currently defined to Dynamic Storage Management System as having the index relative to /XANØPP/ of the reserved block.

NIDX - ("new IDX") the IDX variable that, on output from DSMS, contains the index value originally contained in the ØIDX parameter.

Description: The DSMS module may be used when the user wishes to redefine an IDX variable. DSMS sets the new IDX variable NIDX to the value of the old IDX variable ØIDX. In addition, word 3 of the reserved block control words is redefined to contain the address relative to /XANØPP/ of the new IDX variable. Therefore, in subsequent DSM operations where the reserved block is relocated due to free storage consolidation the new IDX variable will be updated to contain the new location of the reserved block.

Error Conditions: DSMS aborts with a message if the storage type is invalid, if the user is invalid, or if the IDX is invalid for the reserved block.

3.7.3.10 DSMU - Unlock Dynamic Storage

Purpose: To unlock dynamic storage to permit consolidation of free storage.

Format: CALL DSMU(USER, TYPE)

Arguments:

USER - one to six-character name of user imposing lock against consolidation.

TYPE - three-character code indicating storage type (3HGDS or 3HLDS).

Description : Each time DSMU is called the user lock on the storage type specified is decremented by one. Therefore, if DSML has previously been called more than one time to increment the lock, a single call to DSMU does not insure that the user lock against consolidation of free storage is cleared.

Error Conditions: DSMU aborts with a message if the storage type is invalid or if it is already unlocked.

3.7.3.11 DSMX - Expand a Reserved Block

Purpose: To expand the size of a reserved block, if a free block of the required size is below and adjacent, or if there is an available free block in the free storage chain that will satisfy the increased size of the expanded block.

Format: CALL DSMX(USER, TYPE, IDX, MINI, MAXI, LEN)

Arguments:

USER - one to six-character name of user requesting expansion.

TYPE - three-character code indicating storage area where block resides (3HGDS or 3HLDS).

IDX - address relative to /XANØPP/ of block to be expanded.

MINI - minimum increment of additional storage required by user (must be greater than zero).

MAXI - maximum increment of additional storage required by user (must be greater than or equal to MINI).

LEN - (OUTPUT) new length of reserved block after expansion. This parameter is set to zero if expansion was not accomplished.

Description: The DSMX module is called to increase the size of a reserved block previously reserved via the DSMG module. If there is a free block adjacent and immediately following the reserved block, then the free block is examined for the minimum/maximum increment. If the free block satisfies minimum or maximum increment, or a value between the two, then the reserved block is expanded into the free block below.

If all of the free block below is required to expand the reserved block, then the free block is removed from the free storage chain and the reserved block trailer word is moved to account for the increased size of the block. In addition, the reserved block size contained in the header and trailer control words is incremented appropriately.

If only a portion of the free block below the reserved block is required for expansion, then the free block is reduced in size but remains in the free storage chain. Control words for the expanded reserved block and the reduced free block are adjusted accordingly. In addition, forward and backward pointers in the free storage chain are adjusted according to the new start location of the free block.

If there is no free block following the reserved block, or if the free block below and adjacent is insufficient to satisfy either minimum or maximum increment, then the storage area is searched for a free block sufficient in size to hold the expanded reserve block. If such a free block is found to satisfy minimum or maximum or a value between the two, then the reserved block is relocated and expanded into the free block. The space allocated for the original reserved block is then freed via DSMF. The index contained in the user's IDX variable is adjusted to reflect the reserved block's new location.

If all attempts to expand the reserved block should fail, then the length parameter LEN is set to zero. Otherwise, on return from DSMX, the LEN parameter contains the new expanded size of the reserved block.

Error Conditions: DSMX aborts with a message if the minimum increment exceeds the maximum increment, or if either the minimum or maximum increment is negative or zero.

3.7.4 Auxiliary Modules

A DSM error processor may be called by any one of the DSM modules if an error condition is encountered during its execution.

3.7.4.1 DSM Error Message Writer (DSMERR)

Subroutine DSMERR (NUM, IVAL1, IVAL2) processes fatal and non-fatal DSM errors. NUM, the integer number of the error message to be printed, is negative if the error is fatal and positive if the error is non-fatal. DSMERR prints the informative error message (indicated by the absolute value of NUM) with the specific value(s) causing the error condition (indicated by the input values IVAL1, IVAL2). If the error is fatal, ANOPP is aborted by a call to XEXIT. If the error is non-fatal, a traceback is performed to the user callable DSM module that was active when DSMERR was called.

3.7.5 Hierarchy Charts

A hierarchy chart is a graphical representation of the logical relationships between modules. Figures 1-9 are the hierarchy charts for the DSM modules and the auxiliary module.

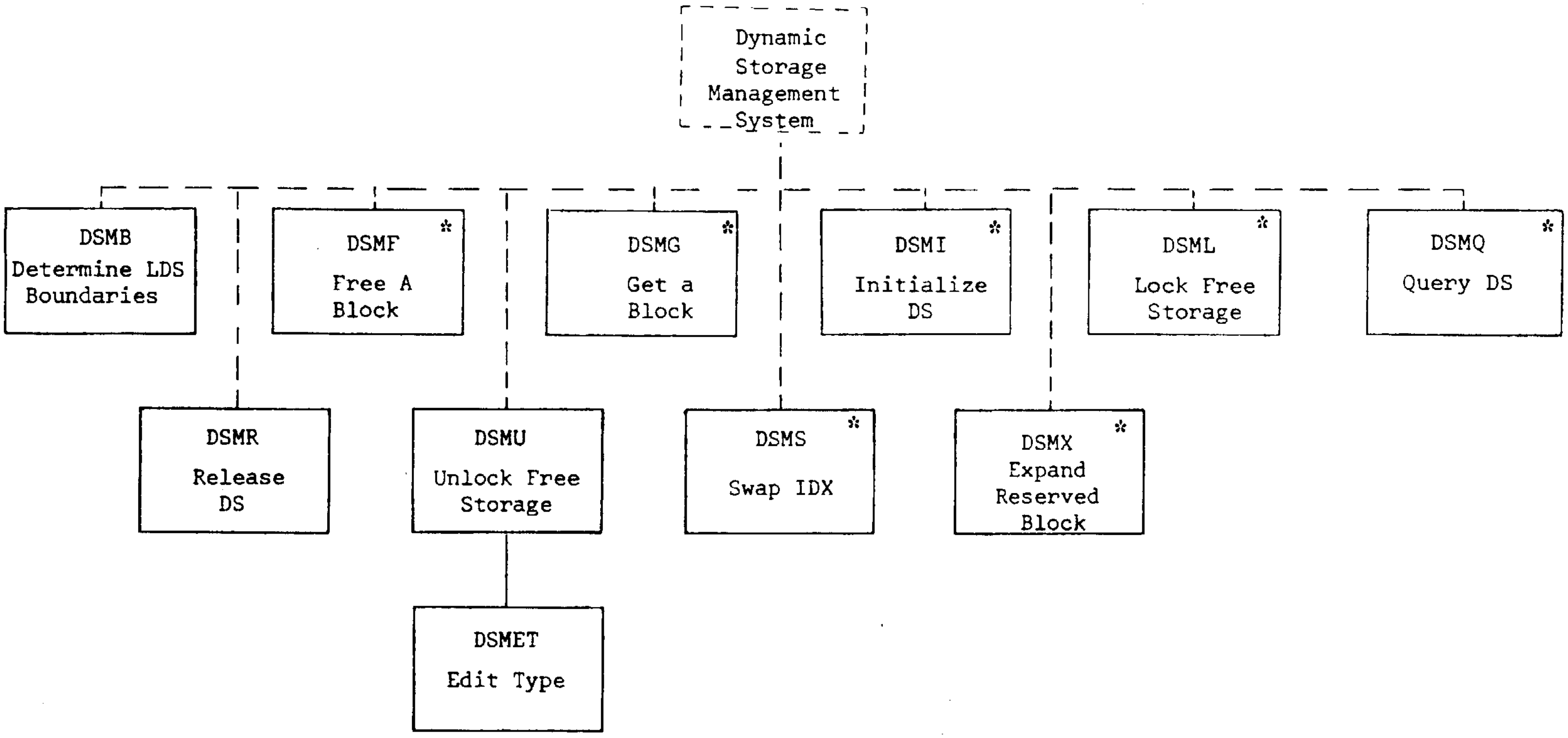
In general, only DSM modules appear as a block entity in the charts and all DSM modules appear at least once. The charts are in alphabetical order with respect to module name except for Figure 1 which represents the logical grouping of the DSM modules. A hierarchy chart for the auxiliary is also among the alphabetized charts.

A module which is not part of DSM but is called by a DSM module is not shown as a block entity, but is listed at the bottom of the chart. The module may be an ANOPP executive module which is a General Utility or a subprogram provided by one of the CDC operating system libraries. In either case, the module is of a utility nature and may be called many times by various DSM modules.

DYNAMIC STORAGE MANAGEMENT SYSTEM (DSM)

Symbols and headings used in the hierarchy charts are listed below:

NAME purpose	NAME - module name purpose - brief description
NAME	Represents logical module not existing as entity. It is used for logical grouping.
————	indicates lower level module is called by higher level module.
----------	implies logical group with no direct relationship
*	in upper right corner of module block indicates module is expanded as a separate hierarchy.
ANOPP Modules Called:	a list of General Utility Modules called by the module in this figure.
CDC System Library Subprograms Called:	a list of subprograms called by the module in the figure and which are not part of ANOPP but are provided by CDC NOS operating system libraries.



ANOPP Modules Called:

DSMERR, ILØC

CDC System Library Subprogram Called:

IABS

Figure 1. Dynamic Storage Management System Hierarchy Chart

DSMERR

Print Error Message

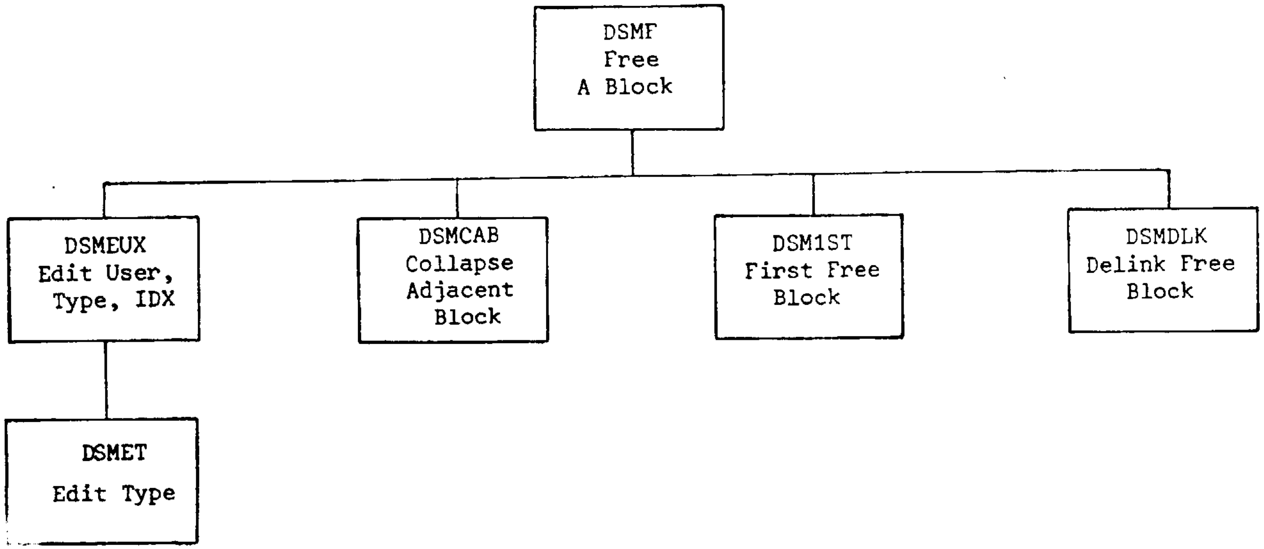
ANOPP Modules Called:

XEXIT, XPLINE, XTRACE

CDC System Library Subprograms Called:

IABS

Figure 2. DSMERR Hierarchy Chart



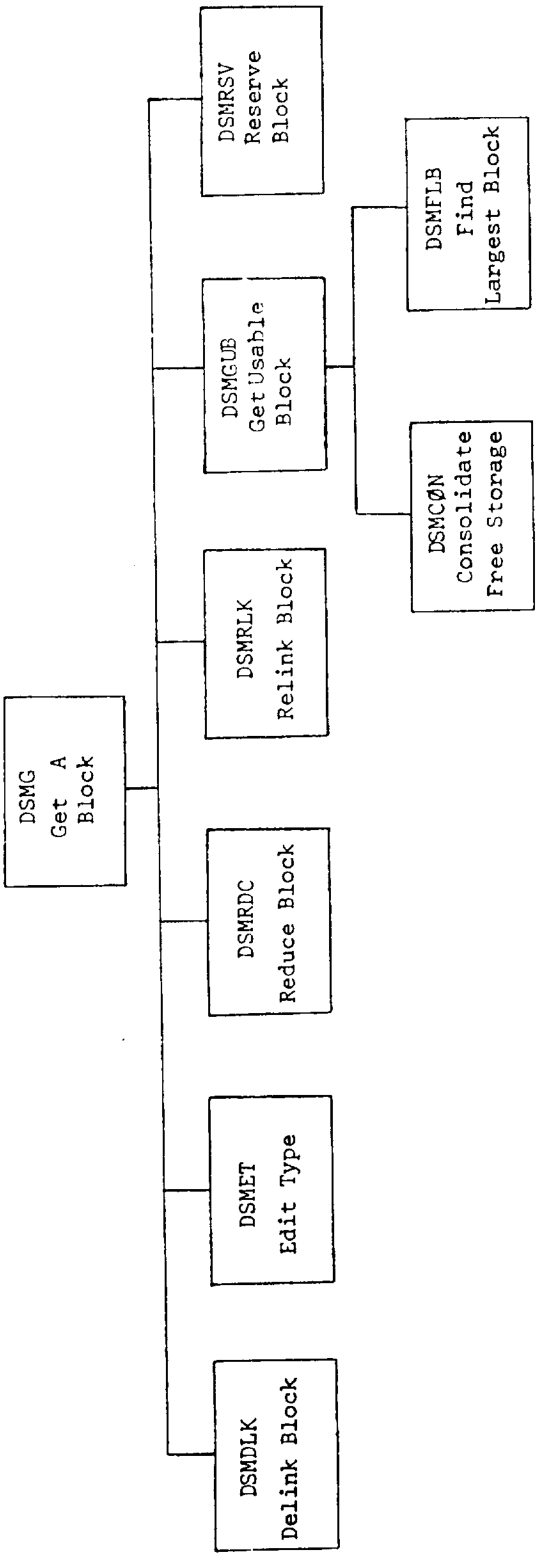
ANOPP Modules Called:

DSMERR, ILØC

CDC System Library Subprogram Called:

IABS

Figure 3. DSMF Hierarchy Chart



ANØPP Modules Called:
DSMERR, ILØC

Figure 4. DSMG Hierarchy Chart

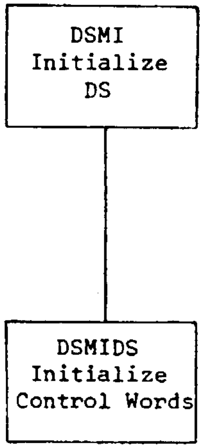
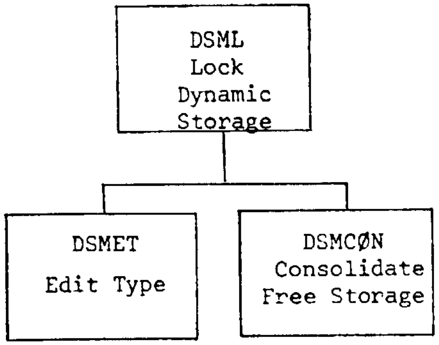


Figure 5. DSMI Hierarchy Chart

ANOPP Modules Called:

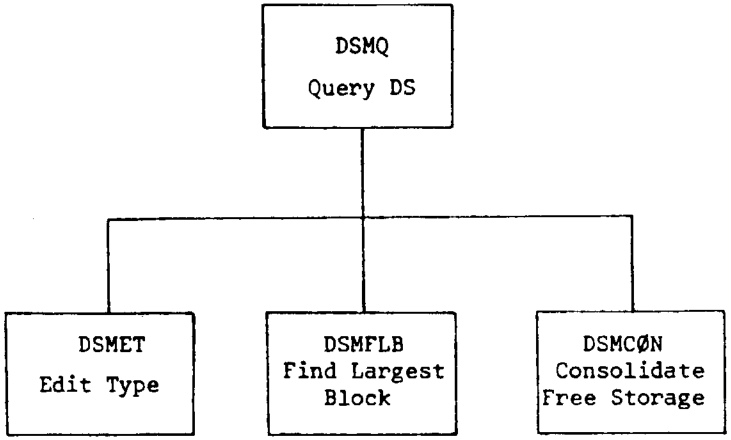
DSMERR



ANOPP Modules Called:

DSMERR

Figure 6. DSML Hierarchy Chart



ANOPP Modules Called:

DSMERR

Figure 7. DSMQ Hierarchy Chart

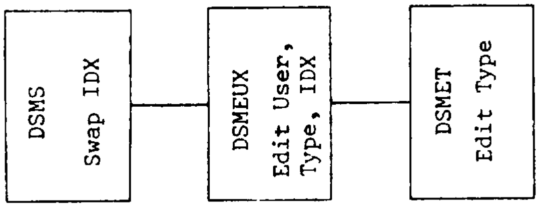
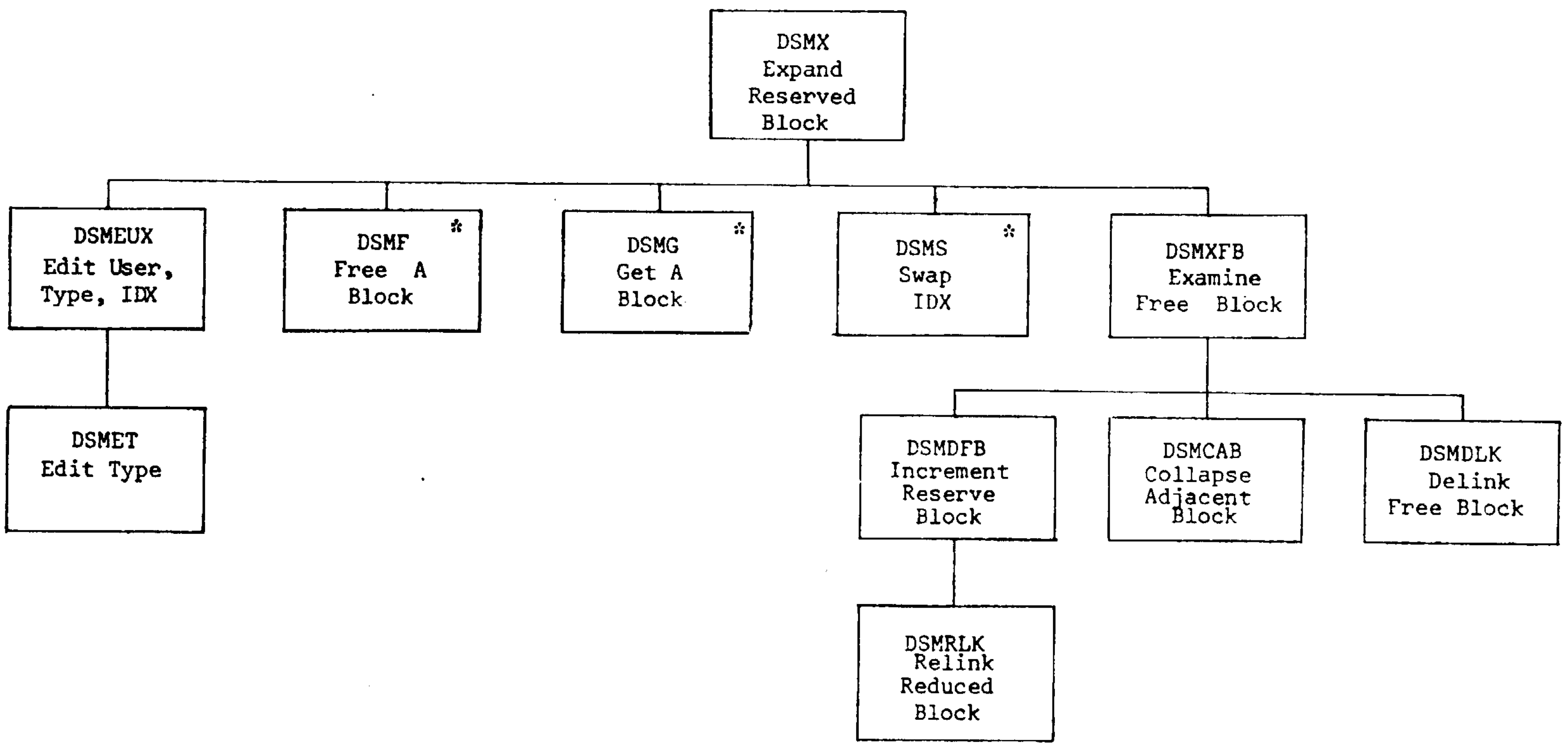


Figure 8. DSMS Hierarchy Chart

ANOPP Modules Called:
DSMERR, ILØC



ANOPP Modules Called:

DSMERR, ILØC,

CDC System Library Subprograms Called:

IABS

Figure 9. DSMX Hierarchy Chart

3.8 UPDATE

3.8.1 Overview

The ANOPP UPDATE processing phase is invoked by module XCSP during the Control Statement Processing Phase (Section 3.5) when an UPDATE Control Statement (CS) is encountered in the primary or secondary input stream. The UPDATE processor provides a method of building the members of a data unit by revising existing members of other data units or by creating new members. The UPDATE CS provides information about the data unit to be changed (ØLDU, if applicable), the data unit to be written (NEWU), the source of the UPDATE directives (SØURCE), and other processing information. UPDATE directives may reside on an existing user data unit/data member (SØURCE = DU(DM) on the UPDATE CS), in card image format or they may follow the UPDATE CS in the primary input stream (SØURCE = * on the UPDATE CS). If the directives follow in the input stream, they are placed on a Uxxx member (Section 3.4.5) on the EM system scratch unit XSUNIT by module XRT during the primary edit phase. The UPDATE processor determines where the directives reside (user member or Uxxx member) and processes them in two major phases, the editing and reformatting phase and the execution phase.

In the editing and reformatting phase, records are read sequentially from the data member defined by SØURCE until a directive is complete (a $ encountered). The entire set of directives is edited, although reformatting ceases when an error is encountered. The steps in editing and reformatting a directive are as follows:

1. The directive is "cracked" by the XCR module to produce a table which includes every field comprising the directive preceded by an integer type code.

2. The cracked directive is then checked for syntax. If no syntactical errors are found, the card image(s) comprising the directive, together with the cracked directive table, is written as one record (a "reformatted" directive) onto the EM scratch unit XSUNIT, data member UPDATS.

3. If the directive is followed by data records (i.e., -ADDR ØLDM=* or -INSERT FRØM=*), the data records are copied, in card image format, onto the data unit XSUNIT, data member UPDATS, followed by a null record written to that member.

4. If the directive is the last record level directive in a set of record level directives, a null record is written following the reformatted record level directive on data unit XSUNIT, data member UPDATS.

EM system control is returned to the module XCSP with the EM system logical error flag, NERR, set to .TRUE. if errors are encountered during this phase.

If no errors are encountered during the editing and reformatting phase, then the execution of the directives phase is invoked. In this phase, the UPDATE module, XUP, sequentially accesses the reformatted directives from data unit XSUNIT, data member UPDATS, and processes them in a single pass performing the functions indicated by each directive.

When either the last directive has been processed or a processing error is encountered, control is returned to the module, XCSP, with NERR set accordingly, and the control statement processing phase continues.

3.8.2 Control Statement

Purpose: The UPDATE Control Statement (CS) provides a means of building a data unit by using an existing data unit as a basis for modifications or by adding members from various sources with no one data unit as a basis or a combination of both. There are two UPDATE modes, Revise and Create, depending on the presence of a data unit as a basis.

The UPDATE CS, present in the primary or Secondary Input Stream, supplies required information for the subsequent processing and allows for selection of various options available to the user. The building of the new data unit is controlled by a set of UPDATE directives which may either immediately follow the UPDATE CS in the Primary Input Stream (Secondary Input Stream prohibited) or be contained in card image record format on a data member.

There are two types of UPDATE directives, member level and record level. Member level directives reference a data member to be processed and include -CØPY, -ØMIT, -ADDR, and -CHANGE. Record level directives must follow a member level -CHANGE directive and reference a particular record(s) to be processed. Record level directives include -INSERT, -DELETE, and -QUIT.

The Revise UPDATE mode is invoked when an existing data unit is to provide a basis for modification. The ØLDU parameter on the UPDATE CS specifies the basis data unit.

The -CØPY and -ØMIT directives are valid only during Revise mode and they reference data members on the ØLDU data unit. All other directives are also valid during a Revise mode UPDATE.

The Create UPDATE mode is invoked when there is no existing data unit to provide a primary basis for modification. The ØLDU parameter on the UPDATE CS is omitted. All data members to be written to the new data unit (with or without modification) will come from various data units known to the ANOPP run. The -ØMIT and -CØPY directives are invalid. The -ADDR and -CHANGE directives are valid.

No data unit utilized by UPDATE (except the new data unit being built) is altered as a result of the UPDATE process. The data members residing on the units are sources of reference only. However the new data unit is always altered as a result of UPDATE. At the start of UPDATE processing the new data unit is "wiped clean" of any members which currently reside on the unit (unless the unit has been write-protected via an ARCHIVE CS in which case an error condition is invoked and UPDATE in inhibited).

Format:

[label]   UPDATE   [ØLDU = du_1,]   NEWU = du_2,   [ALL,]
SØURCE = $\{\overset{*}{d}u_3\ (dm_3)\}$ [LIST = x [x] ...] $

label - (optional) label name

ØLDU clause - (optional) the presence of the ØLDU clause indicates a Revise mode UPDATE. The data unit name specified by du_1, will be used as the basis for UPDATE processing. Member level and record level directives which allow a default of ØLDU data unit or imply the ØLDU data unit, will use du_1, as the required unit. The omission of the ØLDU clause indicates a Create mode UPDATE. Defaults or implications of an ØLDU data unit on all directives are invalid in Create mode.

NEWU clause - du_2 is the name of the data unit to be built during UPDATE processing. It must be known to Member Manager (see Create and Attach CS) and must not be write-protected (see ARCHIVE CS). All members existing on du_2 will be erased (i.e., du_2 will be wiped clean) as if the du_2 had been created via the CREATE CS.

ALL - (optional) if present, indicates a full update of the basis data unit (ØLDU parameter) is to be performed. All data members on the ØLDU data unit which are not processed by a member level directive will be copied to the NEWU data unit. A member will not be copied by ALL if the member name is the same as any of the following:

1. The dm name specifying a member on ØLDU in the ØLDM clause of the -ADDR or -CHANGE directive.
2. The ndm name on the -ADD or -CHANGE directive (either explicitly stated via the NEWM clause or implied by its omission).
3. A dm name on the -CØPY or -ØMIT directive.

If ALL is omitted the full update will not be performed for a Revise mode UPDATE.

The ALL field has no meaning on a Create mode UPDATE.

SØURCE clause - the SØURCE clause specifies where the set of UPDATE input directives will be found. * indicates the directives immediately follow in the Primary Input Stream. The use of * is not valid if the UPDATE CS appears in a Secondary Input Stream (i.e., processed as a result of a CALL CS). The set of UPDATE directives are terminated by an END* CS.

du_3 (dm_3) - specifies the data unit and member containing the directives. dm_3 must contain card image records with 10A8 format (dm_3 may have been built previously via a DATA CS).

LIST clause - (optional) if present, the LIST clause specifies the type of printed output required from the UPDATE processing.

The list following the = is a sequence of letters specifying the sections of output desired. The list may be a combination of the following:

E - Directive Echo Section
S - Summary Section
C - CHANGE Members Section
A - ADDR Members Section

An additional Header Section is automatically selected. The printed output for the various sections is described in Section 3.8.6.

If the LIST clause is omitted only the Header Section is selected.

Examples:

```
LAB1      UPDATE NEWU  =  U1, SOURCE  =  *, LIST  =  S      $
          (directive set)

END* $

UPDATE    ØLDU  =  U001, NEWU  =  U002, ALL, SØURCE  =  DU5(M1), LIST  =  SEA  $

UPDATE    ØLDU  =  U002, NEWU  =  ABC, SØURCE  =  *    $
          (directive set)

END* $
```

3.8.3 Member Level Directives

3.8.3.1 General Format

The format of a member level directive is shown below:

```
directivename  field [field  ..] $
```

The directivename is -ADDR, -CØPY, -ØMIT, or -CHANGE. A blank separates directivename from the fields which are dependent upon the particular directivename. Fields are separated by a delimiter and may be continued just as in the Control Statement format. The delimiters are the same as those for control statements.

3.8.3.2 -ADDR

Purpose: To add to the NEWU data unit a data member which is found on the ØLDU unit, on a unit other than ØLDU, or on card images in the Primary Input Stream. There are two formats:

Format 1:

```
-ADDR  ØLDM =*, NEWM = ndm  [,FØRMAT = format] [,MNR=n]  $
```

OLDM clause - * indicates the member to be added is in card images immediately following the -ADDR directive. It is terminated when another member level directive is encountered. If the format specified or implied in the FØRMAT clause is not CI then a $ must terminate each record and the next record must start on the next card (i.e., characters following the $ on a card are comment). The $ is not considered part of the record. If the FØRMAT clause is omitted or specified as CI, then each record will consist of one card image (10A8) and a $ is not required for record termination.

NEWM clause - ndm specifies the name of the data member added to the NEWU data unit. It must not be the name of a member on NEWU as a result of a previous directive.

FØRMAT clause - (optional) indicates the format of the member, ndm. Format may be one of three forms:

1. 0 - indicates unformatted.

   The characters on the card(s) will be interpreted and converted as types I, RS, RD, CS, CD, L, and a Hollerith string of type A_i. The converted fields will be written to member ndm as unformatted.

2. $nHc_1...c_{n-1}$ - indicates fixed or variable length format.

   The character string $(c_1...c_{n-1})$ describe the format of the records to be written to ndm. The fields on the cards will be interpreted and converted as types I, RS, RD, CS, CD, L and Hollerith character string type A_i. The correspondence between the type field specified on the format and the card image field is given below. The card image fields are as described for all Control Statements.

format code	description	card image field (S)	number words written to record
I	integer	I	1
RS	real single precision	RS	1
RD	real double precision	RD	2
CS	complex single precision	two successive RS fields	2
CD	complex double precision	two successive RD fields	4

Each element code is separated by a comma. A multiplier may optionally precede parentheses enclosing a single element or a group of elements. The multiplier specifies the number of times the element(s) are to be repeated. The character * may be used as an indefinite multiplier when preceding the last element(s) of the format. The * specifies an indefinite repeat of the element group.

A fixed length format results when the format does not contain the indefinite multiplier (*). Each record will be of the same length determined by the format. A variable length format results when the format includes the indefinite multiplier (*). The records are variable length depending on the number of elements written which may or may not include the indefinite repeat group in whole or in part.

3. 2HCI - indicates the card images following should not be interpreted and converted but instead written directly to member ndm, which will have a card image format.

The formats resulting from 0 or nH--- or 2HCI are identical to Member Manager (MM) formats (except MM requires a terminating $) and the sections describing MM should be referenced for further description. If the FØRMAT clause is omitted, then FØRMAT = 2HCI is assumed.

MNR clause - (optional) indicates the maximum number of records that ndm may contain.

n is an integer specification.

If the clause is omitted, the executive system default for Member Manager is used (10,000 currently).

Example 1:

```
-ADDR ØLDM = *, NEWM = M1, MNR = 3  $
RECORD 1 WILL BE THIS CARD
RECORD 2 WILL BE THIS CARD
RECORD 3 WILL BE THIS CARD
```

```
- (next directive)
```

Result: Member M1 has CI format and contains three records.

Example 2:

```
-ADDR ØLDM = *, NEWM = M2, FØRMAT = 20HI,2RS,A9,RD,CS,L,CD$, MNR = 2  $

1 1.5 2E+01 9HABCDEFGHI 3D-01 1.5 2.0

.TRUE. .5D+00 -.841D-06  $

11 .5 -.696E-110 9H123456789 10.5D+01

.696E+29 .32E+31 .FALSE. .919D+01 .210D+06  $

- (next directive)
```

Result: Member M2 has fixed format (I,2RS,A9,RD,CS,L,CD) and contains two records of length 14.

Example 3:

```
-ADDR ØLDM = *, NEWM = M3, FØRMAT = 0 $

.693 $

.70D+01 .898D+20 .TRUE. 3HABC  $

10   20   30   $

- (next directive)
```

Result: Member M3 is unformatted and can have a maximum of 10,000 records but actually contains three records as follows:

Record 1 is 1 word long (one floating point RS)
Record 2 is 6 words long (two floating point type RD, a logical .TRUE., one word containing string ABC left justified and blank filled)
Record 3 is 3 words long (three integers type I)

Format 2:

-ADDR ØLDM = $\left\{\begin{matrix}\text{du (dm)} \\ \text{dm}\end{matrix}\right\}$,NEWM=ndm $

ØLDM clause - indicates where the member to be added is found. du(dm) specifies the unit and member name. du may or may not be the name of the unit named as the ØLDU unit on the UPDATE CS. If du is omitted, the unit named as ØLDU is assumed. The new ndm member will be identical to the member named in the SØURCE clause. dm and ndm may or may not be the same name.

NEWM clause - (optional) ndm is the name of the data member added to the NEWU. If the clause is omitted the name dm specified in the ØLDM clause is used for ndm.

Examples:

```
-ADDR  ØLDM = UNIT1 (MEMA), NEWM=M1  $
-ADDR  ØLDM =    MEM5, NEWM=MZ  $
-ADDR  ØLDM  =    M3  $
```

3.8.3.3 -CØPY

Purpose: To copy to the NEWU data unit one or several data members on the ØLDU data unit.

Format:

```
-CØPY dm [,dm   ...] $
```

dm - name of the data member residing on ØLDU to be copied to NEWU. dm must not be the name of a member already written to NEWU as a result of a previous member level directive.

Examples:

```
-CØPY  ABC, M1, MEMB, D1  $
-CØPY  XYZ  $
```

3.8.3.4 -ØMIT

Purpose: To omit the copying of one or several data members on the ØLDU data unit to the NEWU data unit during processing of the ALL UPDATE CS option.

Format:

```
-ØMIT   dm [,dm   ...] $
```

dm - name of the data member residing on ØLDU to be omitted as a copy possibility during ALL processing.

Examples:

```
-ØMIT  ABC, MEM1, MEM2, MEM5  $
-ØMIT  MEM10  $
```

3.8.3.5 -CHANGE

Purpose: To change a data member on either the ØLDU data unit or another data unit via record level directives which follow the -CHANGE directive and to write it on the NEWU data unit with rename capability.

Format:

-CHANGE ØLDM= $\left\{ \begin{matrix} \text{du(dm)} \\ \text{dm} \end{matrix} \right\}$ [,NEWM=ndm] [,MNR=n] $

ØLDM clause - specifies the data member to be changed.

du(dm) - specifies the data member to be changed. Data unit du may be the unit named as ØLDU on the UPDATE CS. dm with du omitted specifies the data member which resides on the ØLDU data unit to be changed. In either case, the data member, dm, is referenced on the record level directives as ØLDM.

NEWM clause - (optional) ndm is the name to be given to the member written to NEWU. If the clause is omitted then the name dm specified in the ØLDM clause is used.

MNR clause - (optional) specifies the maximum number of records the member ndm will contain (regardless of the number of records contained on the ØLDM member). If omitted, the MNR used in creating ØLDM will be used with a possible increment as explained below.

NREC_0 = current number records on ØLDM

MNR_0 = MNR used when ØLDM was created

MNR_{ndm} = MNR to be used in creating ndm.

```
IF   $(\text{NREC}_0 \leq .95 * \text{MNR}_0)$
THEN $\text{MNR}_{\text{ndm}} = \text{MNR}_0$
ELSE $\text{MNR}_{\text{ndm}} = 1.1 * \text{MNR}_0$
```

Usage: Record level directives immediately follow the -CHANGE directive and direct the altering or changing of the ØLDM member. Processing of the -CHANGE terminates when either another member level directive or the end of the input set of directives to the UPDATE CS is encountered.

Examples:

```
-CHANGE ØLDM=M1, NEWM=M2  $
          .
          .
          .
(record level directives)
          .
          .
          .
-(Member level directive)
-CHANGE ØLDM=UN1(MEM),MNR=2000  $
          .
          .
          .
(record level directive)
          .
          .
          .
```

3.8.4 Record Level Directives

3.8.4.1 General Description

The format of a record level directive is shown below:

```
directivename   field   [field   ...] $
```

The directivename is -INSERT, -DELETE, or -QUIT. A blank separates the directivename from the fields which are dependent upon the particular directivename. Fields are separated by a delimiter and may be continued just as in the Control Statement format. Delimiters are the same as for the Control Statement Format.

Throughout record level directives operands i and j are used as pointers to relative record positions in the old member. Record level directives must be processed sequentially with respect to i and j. This is necessary because the new member is created in one pass. So, a directive referencing records 5 through 8 (i_1 through j_1) must be called before a directive referencing records 9 through 10 (i_2 through j_2) and never vice versa. The i value (if present) in any directive must be greater than the i (and j if present) specified in the preceding directive as noted in each of the following directives.

This is accomplished by an old member reference pointer, P, which is initialized to zero at the beginning of the CHANGE processing. The i value specified on a record level directive must be greater than P. Upon completion of the directive P assumes one of the following three values:

1. the j value (if present)
2. the i value (j not present, i present)
3. unchanged (i and j not present)

The next record level directive must have an i value (if present) greater than the new value of P. The i parameter may be omitted on the -INSERT and -QUIT directives as described in the following sections.

The -INSERT, -DELETE, and -QUIT record level directives are processed under control of the preceding -CHANGE member level directive. The -CHANGE directive is in control until another member level directive or the end of the input SOURCE is encountered. If a -QUIT is encountered, processing of the member being CHANGEd is terminated according to the -QUIT specifications. If a -QUIT is not present and the -CHANGE is terminated by encountering a member level directive or the end of the input stream, then the member being CHANGEd is completed as follows. Any records remaining on the old member following the reference pointer P will be copied to the new member. (i.e., records P+1 through the last record on ØLDM are copied to the new member). The new member is then closed.

3.8.4.2 -INSERT

Purpose: To insert one or more records into a data member being changed.

Format:

$$\text{-INSERT}\ \left[\ i\ \right]\left[\text{FRØM=}\left\{\begin{matrix}*\\ \text{ØLDM}\end{matrix}\right\}\right]\left[(m\ [,n])\right]\ \$$$

i - the record after which the specified records will be inserted. i must be greater than the old member record pointer (P). Records P+1 through and including record i will be copied to the new member followed by the inserted records.

The i may be omitted in which case the records are inserted immediately onto the new member and P remains unchanged. Thus records may be inserted at the beginning of the member (when P=0) if -INSERT with i omitted is the first directive following the CHANGE.

FRØM clause - the FRØM clause specifies the source of the records to be inserted. FRØM=* indicates the records to be inserted immediately following the INSERT directive. FRØM=ØLDM indicates the member named as ØLDM on the CHANGE directive contains the records to be inserted. The entire FRØM clause may be omitted, in which case FRØM=* is assumed. The FRØM=* clause is valid in the Primary or Secondary Input Stream.

If the format of ØLDM is unformatted, fixed, or variable then a $ must terminate each input record and the next record must begin on a separate card image.

(m,n) - specifies which records on the ØLDM are to be inserted. It is ignored if FRØM=* is present. Records m through n will be inserted ($n \geq m \geq 1$). n may be omitted and, if so, m = n is assumed. The RECØRDS clause may be omitted and if so, all of the records on ØLDM will be inserted. Records which follow immediately in the input stream are card images. If the format of ØLDM is card image (2HCI used when created via UPDATE ADDR) then each card image is copied directly to ndm as a record.

After processing an -INSERT, the record pointer in the ØLD member points to record i if i is present. If i is not present then the old member record pointer is unchanged.

Example 1:

Assume N1 is a member on the ØLDU data unit and contains 1000 card images, perhaps built via a DATA CS. Then to insert two card images following the sixth record on N1 the following is required:

```
-CHANGE  ØLDM = N1  $
-INSERT  6  $
(card image)
(card image)
-(next member level directive)
```

Example 2:

The following directive copies records 1 through 5 from the old member and inserts them after record 6 of the old member.

```
-INSERT 6, FRØM=ØLDM  (1,5)     $
```

Example 3:

Assume the ØLDU data unit is UN1 and contains member MEM which consists of 2000 records. MEM was created with a maximum record number (MNR) of 2010 records. The new member on NEWU is to be named MEMNEW. To copy records 5 through 10 from MEM to the beginning of the new member and copy (without altering) the other records in MEM, the following sequence may be used.

```
-CHANGE  ØLDM = MEM,  NEWM = MEMNEW,  $
-INSERT  FRØM = ØLDM (5,10)  $
-(next member level directive)
```

Example 4:

Assume the name as Example 3 except records 5 through 20 are to be inserted. This would result in 2016 records on MEMNEW but the automatic expansion algorithm for MNR is sufficient therefore the following is used:

```
-CHANGE  ØLDM = MEM, NEWM = MEMNEW  $
-INSERT  FRØM = ØLDM  (5,20)  $
-(next member level directive)
```

Example 5:

Assume the same as Example 3 except records 5 through 205 are to be inserted. The automatic expansion algorithm will result in 2100 which is insufficient for the MEMNEW of 2201 records. Therefore MNR must be specified.

```
-CHANGE  ØLDM = MEM,  NEWM = MEMNEW,  MNR = 2500  $
```

3.8.4.3 -DELETE

Purpose: To delete records on the member being changed.

Format:

```
-DELETE   i [,j]    $
```

i,j - i and j specify the range of records to be deleted ($P \leq i \leq j$) where P is the old member record pointer. Records P+1 thru i-1 ($p < i$) on ØLDM are copied to the new member. The records i thru j are effectively deleted by setting P to the value of j.

j may be omitted which results in the one record i being deleted (same as i = j).

Example:

-CHANGE ØLDM = JET $	At initialization record pointer P set to zero. (P=0)
-DELETE 4,6 $	Records 1,2,3 copied to new member; 4,5,6 skipped. (P=6)
-INSERT FRØM=ØLDM (12,13) $	Records 12 and 13 are copied from old member onto new member after current position of P which is 6. (P unchanged)
-DELETE 9,10	Records 7 and 8 are copied from old member to new member; 9,10 skipped. (P=10)

Old Member	New Member
Rec 1	Rec 1
Rec 2	Rec 2
Rec 3	Rec 3
Rec 4	Rec 12
Rec 5	Rec 13
Rec 6	Rec 7
Rec 7	Rec 8
Rec 8	
Rec 9	
Rec 10	
Rec 11	
Rec 12	
Rec 13	
Rec 14	

3.8.4.4 -QUIT

Purpose: To terminate processing of the member being altered via the -CHANGE directive after a specified record.

Format:

$$\texttt{-QUIT}\quad\left[\texttt{i}\right]\quad\texttt{\$}$$

i - (optional) specifies the record with which to terminate processing. All records from the current position of the record pointer plus one through record i are copied onto the new member. Record i will be the last record copied to the new member.

If i is omitted processing on the new member is terminated immediately. The -QUIT directive, if present, must be the last record level directive under the control of the -CHANGE directive.

3.8.5 Format Summary

The following is a summary list of valid formats for the UPDATE CS and the UPDATE Member level and Record level directives:

UPDATE CS

$$\left[\texttt{label}\right]\ \texttt{UPDATE}\ \left[\texttt{ØLDU = du}_1\texttt{,}\right]\ \texttt{NEWU = du}_2\texttt{,}\ \left[\texttt{ALL,}\right]\ \texttt{SOURCE =}\begin{Bmatrix}* \\ \texttt{du}_3\texttt{(dm}_3\texttt{)}\end{Bmatrix}\left[\texttt{LIST= x}\ \left[\texttt{x ...}\right]\right]\texttt{\$}$$

Member Level Directives

$$\texttt{-ADDR}\ \ \texttt{ØLDM=*,}\ \ \texttt{NEWM=ndn}\ \left[\texttt{,FØRMAT=}\begin{Bmatrix}\texttt{0} \\ \texttt{nH...\$} \\ \texttt{2HCI\$}\end{Bmatrix}\right]\left[\texttt{,MNR=n}\right]\quad\texttt{\$}$$

$$\texttt{-ADDR}\ \ \texttt{ØLDM=}\begin{Bmatrix}\texttt{du(dm)} \\ \texttt{dm}\end{Bmatrix}\left[\texttt{,NEWM=ndm}\right]\ \texttt{\$}$$

$$\texttt{-CØPY}\ \ \texttt{dm}\ \left[\texttt{,dm}\quad\texttt{...}\right]\texttt{\$}$$

$$\texttt{-ØMIT}\ \ \texttt{dm}\ \left[\texttt{,dm}\quad\texttt{...}\right]\texttt{\$}$$

$$\texttt{-CHANGE}\ \ \texttt{ØLDM=}\begin{Bmatrix}\texttt{du(dm)} \\ \texttt{dm}\end{Bmatrix}\left[\texttt{,NEWM=ndm}\right]\ \left[\texttt{,MNR=n}\right]\ \texttt{\$}$$

Record Level Directives

```
-INSERT [i] [FRØM= { *    } ] [(m,n)]    $
                   { ØLDM }
-DELETE  i  [,j]   $
-QUIT  [i]     $
```

3.8.6 UPDATE Output Description

3.8.6.1 HEADER SECTION

```
UPDATE PROCESSING BEGINNING WITH THE FOLLOWING PARAMETERS

  { CREATE }
  {        }   MODE     New Data Unit = NAME (A8)     Old Data Unit = NAME (A8)
  { REVISE }

                                       { PRIMARY INPUT STREAM          LIST = S E A C
       SOURCE OF UPDATED DIRECTIVES IS {
                                       { DATA UNIT NAME (A8), DATA MEMBER NAME (A8)
```

3.8.6.2 DIRECTIVE ECHO SECTION

(all card image directives on SØURCE data member or * are listed in 10A8 format)

```
                                   .
       EDITED CARD IMAGE           .
                                   .
```

NOTE - entire Directive Echo is printed prior to any processing.

3.8.6.3 SUMMARY SECTION

```
                              UPDATE PROCESSING SUMMARY

                            NEW DATA UNIT  =  NAME (A8)

MEMBER              NUMBER OF           RECORD TYPE         MAXIMUM        RESULT  OF
 NAME                RECORDS                                LENGTH

Name (A8)             (I3)                  CI              (I14)        ( -ADDR   )
                                                                         { -CHANGE }
                                                                         { -CØPY   }
                                                                         ( -ALL    )

TOTAL OF (I5) MEMBERS ON NEW DATA UNIT
```

3.8.6.4 CHANGE MEMBER SECTION

```
DATA UNIT  =  NAME (A8)    DATA MEMBER  =  NAME (A8)        FORMAT  = { FORMAT IMAGE }
                                                                      { UNFORMATTED  }

                              RECORD

                                1       card image record (10A8) if CI

                                                          OR

                                        octal record (5Ø23) multiple

                                .

                                .

                                .

                                n

                              (I7)      one line of (10A8) or multiple lines of 5Ø23.
```

NOTE: The member written to the new unit is listed upon completion of the CHANGE Directive.

3.8.6.5 ADDR MEMBER SECTION

```
                                                                     { FORMAT IMAGE }
DATA UNIT  =  NAME (A8)      DATA MEMBER  =  NAME (A8)      FORMAT = { UNFORMATTED  }

                      RECORD

                        1       card image record (10A8) if CI

                                           OR

                                octal record (5Ø23) multiple print lines

                        .

                        .

                        .

                        n

                       (I7)     one line (10A8) or multiple lines (5Ø23)
```

NOTE: The member is listed upon completion of the ADDR Directive.

3.8.7 Error Philosophy

The set of UPDATE member level directives, contained either on the SOURCE data member or in the Primary Input Stream, is executed sequentially beginning with the first directive. If no error occurs in the sequential processing, UPDATE terminates normally when execution of the last member level directive in the directive set is complete. However, if an error is encountered while processing any member level directive or any record level directive under the control of a member level directive (i.e., -CHANGE), then further UPDATE processing is inhibited and UPDATE is terminated.

Upon UPDATE error termination the NEWU unit contains all members written as a result of the previous member level directive which executed successfully. If the error occurred during execution of a record level directive and the member has been partially written as a result of immediately preceding successful record level directives then the partially complete member is considered complete and will be present on NEWU. A partially complete member is the result of the -CHANGE directive being terminated by a record level directive error. The contents of the NEWU unit will be reflected in the summary section which is printed upon error termination if selected via the LIST field on the UPDATE CS.

Errors may be detected in UPDATE processing during one of two phases, the edit phase or the execution phase.

During the edit phase, the set of UPDATE member level or record level directives contained either on the SØURCE data member or in the Primary Input Stream is edited for syntactical errors. The directives are read and checked sequentially, beginning with the first directive in the set. If a directive is syntactically correct, it is then stored in an executable format, known internally to UPDATE. If no error is detected on any UPDATE directive during the edit phase, the entire reformatted set of directives will be input to the execution phase.

If an error is detected on an UPDATE directive during the edit phase, the remainder of the directives in the set will be checked for syntax, but will not be reformatted for execution. The directive(s) in error will be printed, and UPDATE will terminate following the edit phase if any error in syntax is present in the set of directives. The following errors include conditions which would result in an edit phase error and inhibition of execution processing.

1. Required field missing - for example, the ØLDM clause is not present on an -ADDR directive.

2. Invalid field type - for example, on the -ADDR directive the FØRMAT clause is present and contains FØRMAT = 10A8.

3. Incomplete directive - for example, -ØMIT $ which contains no dm fields. This is similar to required field missing.

4. Invalid directive name - for example, a mispunch resulted in the directive -CØPI DM $.

During the execution phase, the reformatted set of syntactically correct UPDATE directives are executed sequentially beginning with the first directive. If no error occurs in the sequential processing, UPDATE terminates normally when execution of the member level directive in the directive set is complete. However, if an error is encountered while processing any member level directive or any record level directive under the control of a member level directive (i.e., -CHANGE) then further UPDATE processing is inhibited and UPDATE is terminated.

Upon UPDATE error termination the NEWU unit contains all members written as a result of the previous member level directive which executed successfully. If the error occurred during execution of a record level directive and the member has been partially written as a result of immediately preceding successful record level directives, then the partially complete member is considered complete and will be present on NEWU. A partially complete member is the result of the -CHANGE directive being terminated by a record level directive error. The contents of the NEWU unit will be reflected in the summary section which is printed upon error termination if selected via the LIST field on the UPDATE CS.

The types of errors which may cause UPDATE error termination are varied. Some error types are specific to a particular directive being executed while other error types are general and apply to all directives. Directive errors are implied by the requirements for each particular directive. The general errors include duplicate member attempted on

NEWU. This error is emphasized and provides the basis for handling conflicting directives. The general conflict rule for executing successfully the set of UPDATE directive is as follows: "there shall not be an attempt to write on the NEWU unit a data member which has the same name as a data member previously written on the NEWU." There is no restriction as to how many times a particular du(dm) may be used on the various directives as long as the resulting member to be written on the NEWU is not the name of a member already residing on the NEWU. UPDATE will not attempt to overwrite members on NEWU. Since NEWU either contains no members upon entry to the UPDATE CS processing phase or is "wiped clean" before UPDATE directive processing begins, the conflict rule is violated only by an incompatible set of directives and not because of the contents of NEWU upon entry to UPDATE processing. The following directive sequence is compatible since the -ØMIT directive does not result in a member being written to NEWU. The -ØMIT directive has no meaning, however since the -ADDR directive would eliminate the possibility of DM2 (if present on the designated ØLDU) being written to the NEWU during ALL processing.

```
-ØMIT  DM2  $
-ADDR  SØURCE = *,  NEWM = DM2  $
```

Also compatible is the following sequence:

```
-ØMIT  DM1  $
-CØPY  DM1, DM3  $
-ØMIT  DM3  $
-ADDR  SØURCE = DM6,  NEWM = DM7  $
-CØPY  DM6  $
-CHANGE  ØLDM = DM6,  NEWM = DM8  $
```

The following sequence results in a conflict error:

```
-CHANGE ØLDM = DM3,  NEWM = DM10
-CØPY  DM10  $
```

3.8.8 Auxiliary Module

An auxiliary module performs a function common to UPDATE modules and is available for use only by UPDATE modules.

3.8.8.1 UPDATE Error Message Writer (XUPERR)

Subroutine XUPERR (NM, CNAME, VAR1, VAR2) processes fatal and non-fatal errors for the UPDATE modules. NM, the integer number of the error message to be printed is negative if the error is fatal and positive if the error is non-fatal. XUPERR prints the informative error message indicated by the absolute value of NM with the name of the calling module (CNAME) and specific value(s) involved in the error condition (VAR1, VAR2). If the error is fatal, ANOPP is aborted by a call to XEXIT.

3.8.9 Hierarchy Charts

A hierarchy chart is a graphical representation of the logical relationship between modules. Figures 1 through 6 are the hierarchy charts for UPDATE.

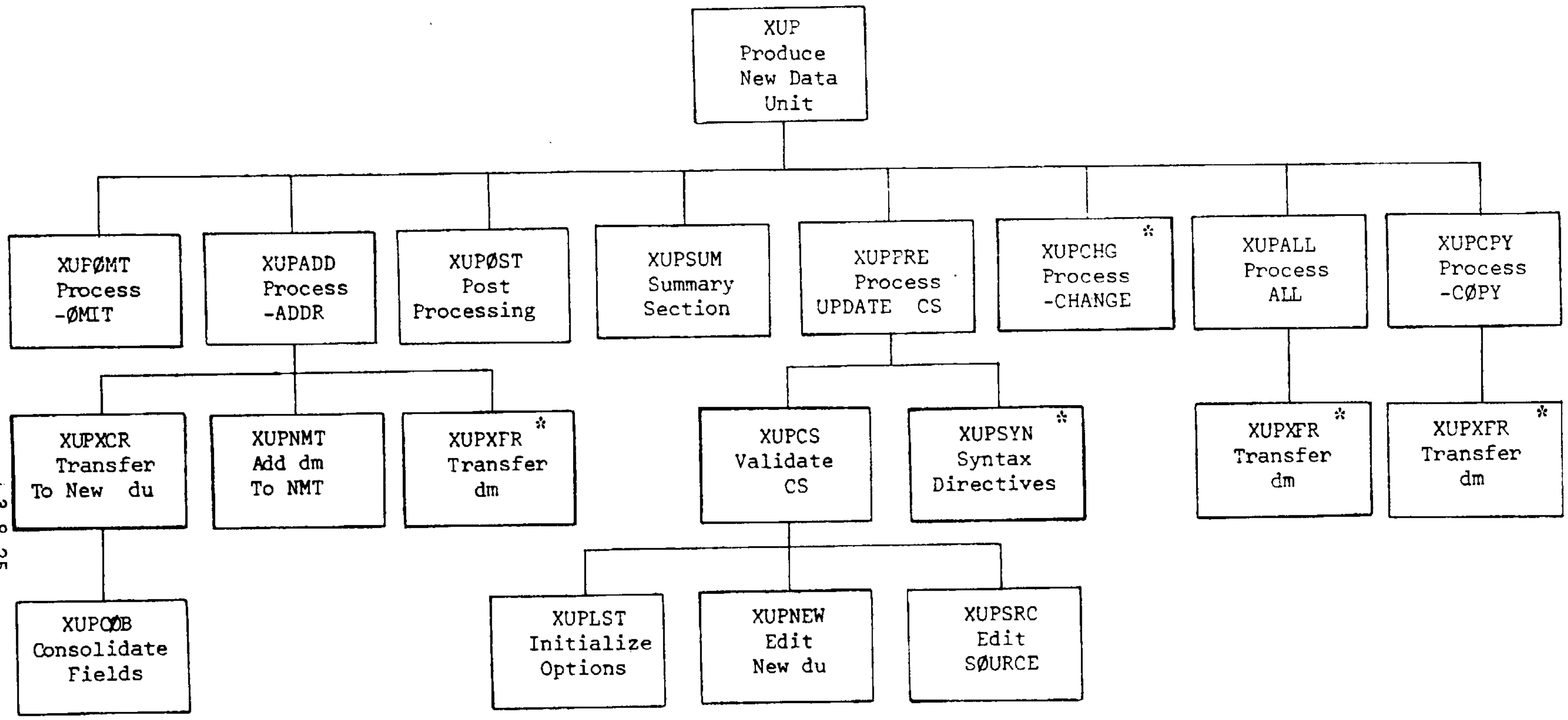
In general, only UPDATE modules appear as a block entity in the charts and all UPDATE modules appear at least once. A module which is not part of UPDATE but is called by an UPDATE module is not shown as a block entity but is listed at the bottom of the chart. The module will be an ANOPP executive module which is part of the Data Base Management System (DBM), the Dynamic Storage Management System (DSM), or the General Utilities, is of a service or utility nature, and may be called many times by various UPDATE modules.

Symbols and headings used in the hierarchy charts are given below:

NAME purpose	NAME - module name purpose - brief description
————	indicates lower module is called by the higher module.
*	in upper right corner of module block indicates module is expanded as a separate hierarchy.

ANOPP Modules Called: a list of DBM, DSM, and General Utility modules called by the modules in this figure.

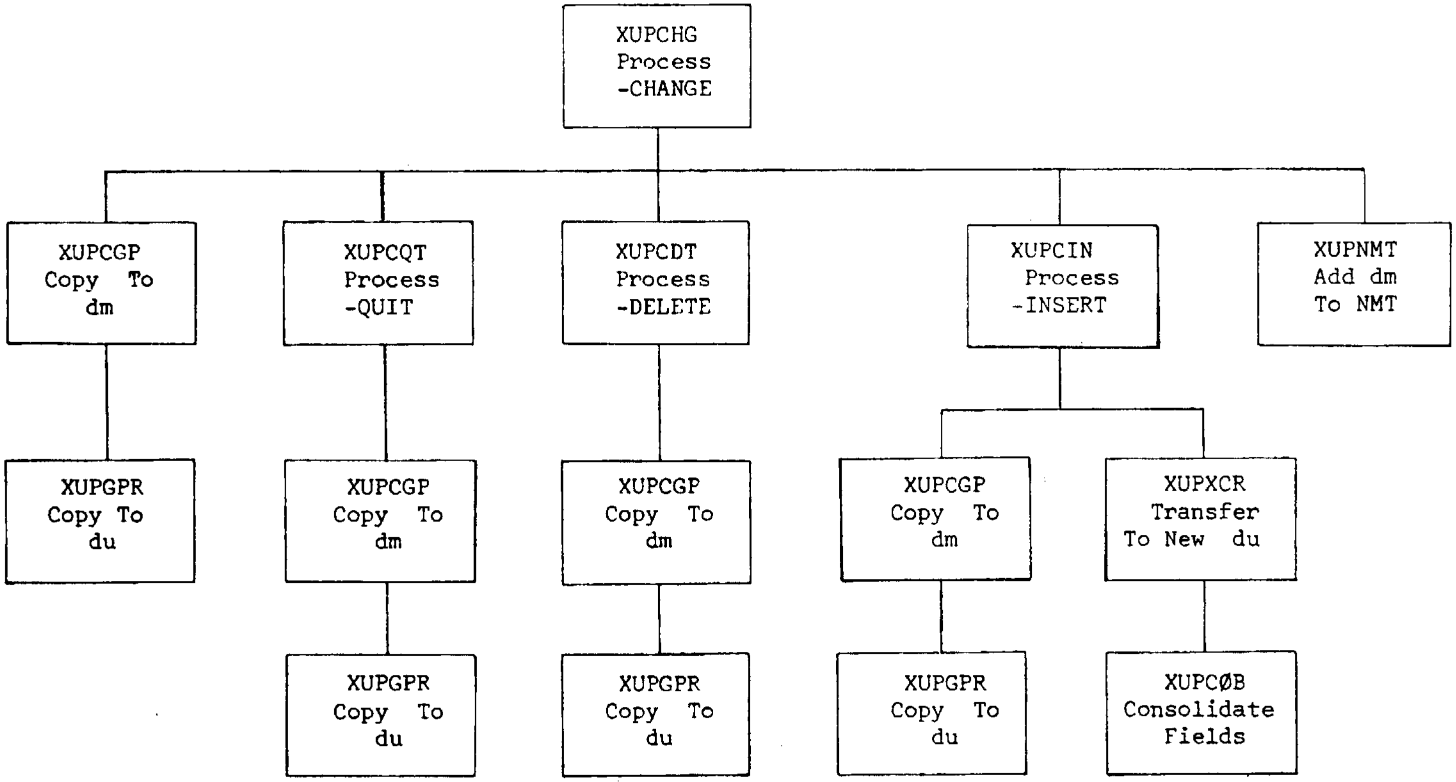
CDC System Library Subprograms Called: list of subprograms called by the module in the figure and which are not part of ANOPP but are provided by CDC NOS operating system libraries.



ANOPP Modules Called:

DSMF, DSMG, DSML, DSMU, DSMX,
MMCLØS, MMDØMC, MMEDNM, MMFEFB, MMGEFB,
MMGETR, MMIØMC, MMØPRD, MMØPWD, MMPUTR,
MMPUTW, MMRMD, MMSUD, MMUHMD, MMVUM,
NWDTYP, XCR, XCTBMD, XFMTQ, XMØVE,
XMPRT, XPAGE, XPLAB, XPLABQ, XPLINE,
XT1FV, XUNPK, XUPERR

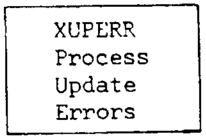
Figure 1. XUP Hierarchy Chart



ANOPP Modules Called:

DSMF, DSMG, DSML, DSMU DSMX,
MMCLØS, MMEDNM, MMGETR, MMGETW, MMØPRD,
MMØPWD, MMPØSN, MMPUTR, MMPUTW, MMREW,
NWDTYP, XCR, XFMTQ, XMØVE, XMPRT,
XPLINE, XT1FV, XUPERR

Figure 2. XUPCHG Hierarchy Chart



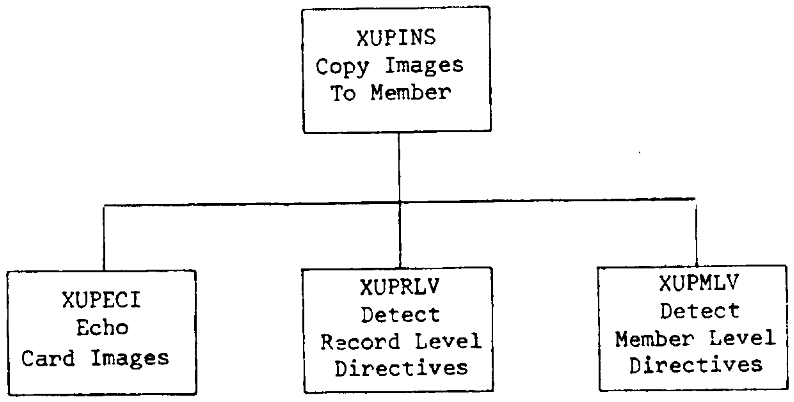
ANOPP Modules Called:

RVALUE, XEXIT, XFETCH, XPLINE

CDC System Library Subprograms Called:

IABS

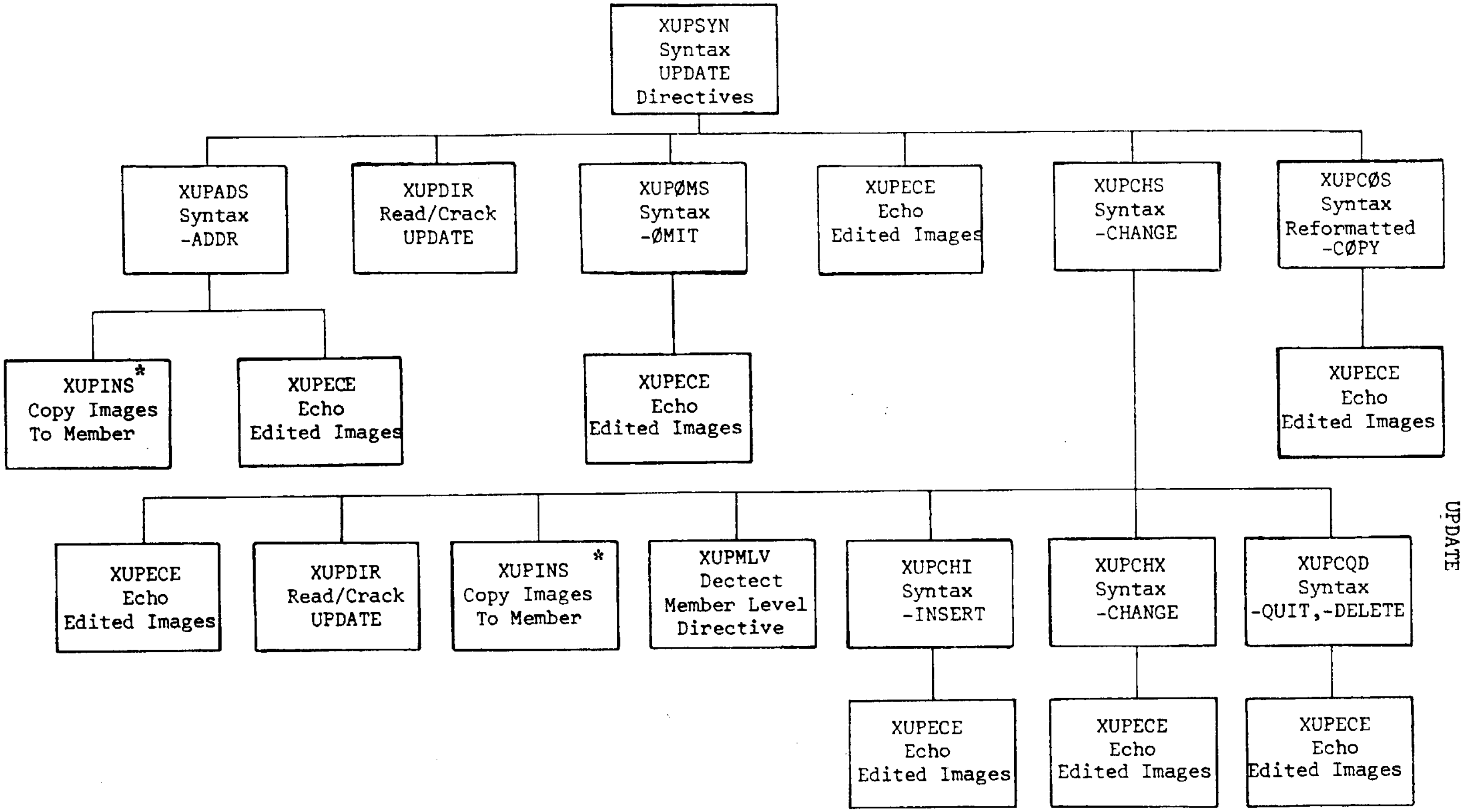
Figure 3. XUPERR Hierarchy Chart



ANOPP Modules Called:

MMGETR, MMPUTR, MMSKIP, XCR, XPLINE

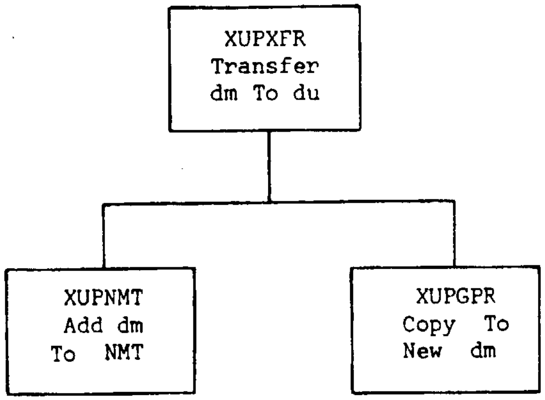
Figure 4. XUPINS Hierarchy Chart



ANOPP Modules Called:

DSMF, DSMG. MMCLØS, MMGETR, MMØPRD,
MMØPWD, MMPUTR, MMSKIP, MMVUM, NWDTYP,
XCR, XPLINE, XUPERR

Figure 5. XUPSYN Hierarchy Chart



ANOPP Modules Called:

DSMF, DSMG, DSML, DSMU, DSMX,
MMCLØS, MMEDNM, MMGETW, MMØPRD, MMØPWD,
MMPUTW, XFMTQ, XT1FV, XUPERR

Figure 6. XUPXFR Hierarchy Chart

3.9 GENERAL UTILITIES

3.9.1 Overview

General Utilities are general purpose subprograms available to all executive system modules. Some resulted during executive system development from the recognition of functions common to several modules. Others are required to replace CDC Fortran intrinsic functions which are NON-ANSI. Most of these utilities are available for use by functional modules. A few subprograms, classed as utilities but not available to the functional modules, are specific to executive system philosophy and design criteria and do not provide the functional module with increased capability.

3.9.2 Reference List

3.9.2.1 ALPHA

Subprogram Type: Logical Function

Calling Sequence: ALPHA(CHAR)

Purpose: Return a function value of .TRUE. if input character, CHAR, is alphabetic. Otherwise, return a function value of .FALSE.

3.9.2.2 DIGIT

Subprogram Type: Logical Function

Calling Sequence: DIGIT(CHAR)

Purpose: Return a function value of .TRUE. if the input character, CHAR, is alphabetic. Otherwise, return a function value of .FALSE. .

3.9.2.3 DVALUE

Subprogram Type: Double Precision Function

Calling Sequence: DVALUE(RS)

Purpose: Enable the use of any mode variable, RS, as if it were double precision with no conversion.

3.9.2.4 IAND

Subprogram Type: Integer Function

Calling Sequence: IAND(I,J)

Purpose: Perform logical product of the two input words I and J.

3.9.2.5 ICD

Subprogram Type: Integer Function

Calling Sequence: ICD(I)

Purpose: Return the integer value which corresponds to the input character I, a valid numeric character (A1) in the range 0-9.

3.9.2.6 ICI

Subprogram Type: Integer Function

Calling Sequence: ICI(I)

Purpose: Return the numeric character (A1) which corresponds to the input variable I, a valid integer in the range 0-9.

3.9.2.7 ICOMPL

Subprogram Type: Integer Function

Calling Sequence: ICOMPL(I)

Purpose: Return the complement of the input variable I.

3.9.2.8 IDATE

Subprogram Type: Subroutine

Calling Sequence: CALL IDATE(D)

Purpose: In the output variable D, return the current date in the A8 format MM/DD/YY.

3.9.2.9 ILOC

Subprogram Type: Integer Function

Calling Sequence: ILOC(I)

Purpose: Return the integer index relative to /XANOPP/, the dimensional array to which all Dynamic Storage addresses are indexed, of input variable I.

3.9.2.10 ILSHFT

Subprogram Type: Integer Function

Calling Sequence: ILSHFT(I,J)

Purpose: Left shift with zero fill the contents of the input word I by J bits. J must have a value in the range 0-number of bits per word.

3.9.2.11 IMASK

Subprogram Type: Integer Function

Calling Sequence: IMASK(I)

Purpose: Form a mask of I high-order bits. I must have a value in the range 0-number of bits per word.

3.9.2.12 IOR

Subprogram Type: Integer Function

Calling Sequence: IOR(I,J)

Purpose: Perform a logical sum of the two input words I and J.

3.9.2.13 IRSHFT

Subprogram Type: Integer Function

Calling Sequence: IRSHFT(I,J)

Purpose: Right shift with zero fill the contents of the input word I by J bits. J must have a value in the range 0-number of bits per word.

3.9.2.14 ISHIFT

Subprogram Type: Integer Function

Calling Sequence: ISHIFT(I,J)

Purpose: Perform a left circular shift (J.GT.0) or right, end-off, sign extend shift (J.LT.0) of the input word I by J bits. The absolute value of J must be less than or equal to the number of bits per word.

3.9.2.15 ITIME

Subprogram Type: Subroutine

Calling Sequence: CALL ITIME(T)

Purpose: In the output variable T, return the time of day in the A8 format hh.mm.ss.

3.9.2.16 IVALUE

Subprogram Type: Integer Function

Calling Sequence: IVALUE(I)

Purpose: Enable the use of any mode word, I, as if it were an integer with no conversion.

3.9.2.17 IXOR

Subprogram Type: Integer Function

Calling Sequence: IXOR(I,J)

Purpose: Perform an exclusive OR of two input words I and J.

3.9.2.18 MEMNUM

Subprogram Type: Integer Function

Calling Sequence: MEMNUM(IN)

Purpose: Convert the three numeric characters in the input word IN to an integer value in the range 0-999. IN is expected to be in the form Axxx (A4) where A is an alphabetic character and xxx are numeric characters in the range 001-999.

3.9.2.19 NUMTYP

Subprogram Type: Integer Function

Calling Sequence: NUMTYP(NAME)

Purpose: Return the integer type code corresponding to the input alpha type code for an ANOPP data type. For a full description of ANOPP Data Types, see the NDTCL array in common block /XCVT/.

3.9.2.20 NWDTYP

Subprogram Type: Integer Function

Calling Sequence: NWDTYP(ITYPE)

Purpose: Return the number of words required for an ANOPP data type given the integer type code. For a full description of ANOPP Data Types, see the NDTCL array in common block /XCVT/.

3.9.2.21 RVALUE

Subprogram Type: Real Function

Calling Sequence: RVALUE(R)

Purpose: Enable use of any mode word, R, as single precision withcut conversion.

3.9.2.22 XASKP

Subprogram Type: Subroutine

Calling Sequence: CALL XASKP(PNAME,ITYPE)

Purpose: Determine if the input variable PNAME, (A8), is a current User Parameter, and, if it is, return the integer type code of the ANOPP data type in the output variable ITYPE. If PNAME is not a current User Parameter, a zero is returned in ITYPE. A User Parameter is a numerical, logical, or character string value established in the control statement stream by a PARAM CS or in a functional module with XPUTP. This value is maintained throughout ANOPP in the User Parameter Table (UPT) and User String Table (UST) and may be changed or retrieved.

3.9.2.23 XBSRIN

Subprogram Type: Subroutine

Calling Sequence: CALL XBSRIN(JXX,JX,NEL,INDEX,IFND,IERR)

Purpose: Performs a binary search for integer Jxx in array Jx.

3.9.2.24 XBSRRD

Subprogram Type: Subroutine

Calling Sequence: CALL XBSRRD(DXX,DX,NEL,INDEX,IFND,IERR)

Purpose: Performs a binary search for real double DXX in array DX.

3.9.2.25 XBSRRS

Subprogram Type: Subroutine

Calling Sequence: CALL XBSRRS(RXX,RX,NEL,INDEX,IFND,IERR)

Purpose: Performs a binary search for real single RXX in array RX.

3.9.2.26 XCR

Subprogram Type: Subroutine

Calling Sequence: CALL XCR(BIN,NC,OUTBUF,LAVAIL,LUSED,ICONT,NBAD,IERR,NF)

Purpose: XCR is the executive crack module which identifies ANOPP Data Types on a card image and converts these fields as required to numerical representations. The converted fields are represented in table form via the OUTBUF array. There is one entry in OUTBUF per field encountered except for blanks and commas which are recognized as delimiters but not entered into the table. The table entries are variable length. The first word of each entry contains the integer type code of the ANOPP data type that follows. The value length is implied by the type code. Fields recognized by XCR for output are Integer, Real Single Precision, Real Double Precision, Hollerith String or Alpha, Logical Operator, Name, Type A Delimiter, and Unrecognizable. The integer type codes and the corresponding value lengths are found in the ANOPP Data Types Table (array NDTCL in common block /XCVT/). The fields as they appear on a card image along with the output integer type codes and value lengths required are shown in Table 1.

ANOPP DATA TYPE	INTEGER TYPE CODE	CARD IMAGE FORM	OUTBUF VALUE FORM	OUTBUF VALUE WORD LENGTH
INTEGER	1	NNNN...N OPTIONAL + - .LE. 18 DIGITS .LE. (2**31)-1	BINARY INTEGER	1
REAL SINGLE PRECISION	2	N. N.NN N.NN+N N.NN-N N.NNEN NEN NE+N NE-N .LE. 14 DIGITS .GE. 10**-293 .LE. 10**+322 OTIONAL + -	BINARY FLOATING POINT	1
REAL DOUBLE PRECISION	3	N.NNDN N.NND+N N.NND-N NDN ND+N ND-N OPTIONAL + - .GE. 10**-293 .LE. 10**+322	BINARY FLOATING POINT	2
HOLLERITH STRING	-N	NHXXX...X N .LE. 132	A8	(N+7)/*
LOGICAL	6	.TRUE. .FALSE.	FORTRAN GENERATED INTEGER	1
ALGEBRAIC OPERATOR	7	+ -	A1	1
LOGICAL OPERATOR	8	.EQ. .LE. .LT. .NE. .GT. .GE.	A4	1
NAME	9	. LE 8 ALPHANUMERIC CHARACTERS FIRST IS ALPHA	A8	1
TYPE A DELIMITER	10	( ) = / *	A1	1
UNRECOGNIZABLE	20+N	NONE OF THE ABOVE OR EXCEED RANGE N=NUMBER CHARACTERS IN FIELD	A8	(N+7)/8

Table 1. Card Images of ANOPP Data Types Recognized by Executive Crack Module XCR.

Continuation calls to XCR may be used to process multiple card images as if they were one contiguous image. In such continuation calls, Hollerith string fields may be continued on the subsequent card image, however, other fields may not be continued.

3.9.2.27 XCRWC

Subprogram Type: Subroutine

Calling Sequence: CALL XCRWC(BIN,NC,OUTBUF,LAVAIL,LUSED,ICONT,LCP,IERR,NF)

Purpose: XCRWC is the executive crack module which identifies fields or multiple card images with variable number of columns and cracks those card images without converting fields. XCRWC builds an output table, OUTBUF, of these fields. There is one entry for each field encountered except for blanks and commas which are recognized as delimiters but not entered into the table. The table entries are variable length. The first word of each entry contains the integer type code of the ANOPP data type that follows. The value length is implied by the type code. Fields recognized by XCRWC are Hollerith String, Type A Delimiter, and Unrecognizable. The integer type codes and the corresponding value lengths are found in the ANOPP Data Type Table (array NDTCL in common block /XCVT/). The fields as they appear on a card image along with the output integer type codes and value length required are shown in Table 2. Continuation calls to XCRWC may be used to process multiple card images as if they were one contiguous image. In such continuation calls, Hollerith string fields may be continued on the subsequent card image, however, other fields may not be continued.

3.9.2.28 XEXIT

Subprogram Type: Subroutine

Calling Sequence: CALL XEXIT

Purpose: Abnormally terminates ANOPP when a fatal error has occurred and performs a trace back from XEXIT to XM. This utility is not available for use by the functional module.

ANOPP DATA TYPE	INTEGER TYPE CODE	CARD IMAGE FORM	OUTBUF VALUE FORM	OUTBUF VALUE WORD LENGTH
Hollerith String	-N	NHxxxx...xx N. LE. 132	A8	(N+7)/8
Type A Delimiter	10	( ) = / *	A1	1
Unrecognizable	20+N	None of the above or exceed range N = number characters in field	A8	(N+7)/8

Table 2. Card Images of ANOPP Data Types Recognized by the Executive Crack Module (XCRWC).

3.9.2.29 XFAN

Subprogram Type: Subroutine

Calling Sequence: CALL XFAN(RNAME,ANAME)

Purpose: XFAN returns ANAME (A8) the alternate name for RNAME (A8) retrieved from the Alternate Names Table (ANT) or RNAME if the desired entry is not in the ANT. Alternate names are established in the control statement stream via the EXECUTE CS. This set of alternate names, maintained in the ANT, is available only during the execution of the functional module.

3.9.2.30 XFETCH

Subprogram Type: Subroutine

Calling Sequence: CALL XFETCH(NAME,VALUE)

Purpose: Fetch the value of the Executive System Parameter specified by NAME (A8). Executive System Parameters currently available for fetching are:

Valid Name	Residence	Type	Description
lWR	/XCVT/	integer	write unit for all FORTRAN write requests to printer. Used by system and functional modules.

3.9.2.31 XFMTQ

Subprogram Type: Subroutine

Calling Sequence: CALL XFMTQ(NAME,ITYP)

Purpose: Returns in ITYP the format type of the member specified by NAME. NAME is a three word array where NAME(1) is the Data Unit name (A8), NAME(2) is the Data Member name (A8), and NAME(3) is to be left unaltered as it is used by Member Manager. Valid values of ITYP are:

CI	- card image format
F	- fixed format
V	- variable format
U	- unformatted

3.9.2.32 XGETP

Subprogram Type: Subroutine

Calling Sequence: CALL XGETP(PNAME,ITYPE,VALUE)

Purpose: Retrieves the value of the User Parameter PNAME (A8) having integer type code ITYPE from the User Parameter Table (UPT) and User String Table (UST). It is assumed that the user has verified that PNAME is in fact an entry in the UPT. A User Parameter is a numerical, logical, or character string value established in the control statement stream by a param CS or in a functional module with XPUTP. This value is maintained throughout ANOPP in the UPT and UST and may be retrieved or changed.

3.9.2.33 XINC

Subprogram Type: Logical Function

Calling Sequence: XINC(IARRAY,RARRAY,DARRAY,IFC)

Purpose: Determines if the input array (IARRAY,RARRAY, or DARRAY), assumed to be in monotonic sequence, is increasing. The array used is determined by the integer type code specified by IFC. Expected type codes are: 1-Integer, 2-Real Single Precision, or 3-Real Double Precision.

3.9.2.34 XMOVE

Subprogram Type: Subroutine

Calling Sequence: CALL XMOVE(FROM,TO,NUM)

Purpose: Moves NUM entries from sending array FROM to the corresponding position in receiving array TO.

3.9.2.35 XMPRT

Subprogram Type: Subroutine

Calling Sequence: CALL XMPRT(NAME)

Purpose: Prints the Data Member specified by NAME, a three word array where NAME(1) is the Data Unit name (A8), NAME(2) is the Data Member name (A8), and NAME(3) is to be left unaltered as it is used by Member Manager.

3.9.2.36 XPAGE

Subprogram Type: Subroutine

Calling Sequence: CALL XPAGE

Purpose: Initializes for printed output a new page with the standard ANOPP header information. The five line ANOPP header follows:

```
Line 1 - MM/DD/YY ANOPP LEVEL N1/N2/N3 PAGE N
Line 2 - Title (16A8)
Line 3 - Subtitle (16A8)
Line 4 - Label (16A8)
Line 5 - blanks
```

where N1, N2, N3 are 1 or 2 digit numbers and N is up to a 6 digit integer.

3.9.2.37 XPK

Subprogram Type: Subroutine

Calling Sequence: CALL XPK(IN,NC,IOUT)

Purpose: Packs NC characters from array IN (A1) into word IOUT. NC is expected to be an integer value between zero and the number of characters per word.

3.9.2.38 XPKM

Subprogram Type: Subroutine

Calling Sequence: CALL XPKM(IN,NC,IOUT,LIOUT)

Purpose: Packs NC characters from array IN (A1) into word array IOUT (A8) and blank fills unused words in IOUT array.

3.9.2.39 XPLAB

Subprogram Type: Subroutine

Calling Sequence: CALL XPLAB(LABEL)

Purpose: Initializes the label line of the ANOPP header for subsequent new page with ANOPP header requests. LABEL is an array (16A8) containing 128 characters.

3.9.2.40 XPLABQ

Subprogram Type: Subroutine

Calling Sequence: CALL XPLABQ(L)

Purpose: Determine current label line of the standard ANOPP header. Output array L (16A8) will contain the current 128 character label line.

3.9.2.41 XPLINE

Subprogram Type: Subroutine

Calling Sequence: CALL XPLINE(LINES)

Purpose: Keeps a running count of lines printed thus far on the current page and detrmines if the number of lines remaining on the current page are sufficient for this print request. If page is not sufficient, a new page is started.

3.9.2.42 XPUTP

Subprogram Type: Subroutine

Calling Sequence: CALL XPUTP(PNAME,ITYPE,VALUE)

Purpose: Establishes or changes a User Parameter value in the User Parameter Table (UPT) or User String Table (UST). A User Parameter is a numerical, logical, or character string value which is maintained in the UPT or UST throughout ANOPP and may be changed or retrieved.

3.9.2.43 XSORTF

Subprogram Type: Subroutine

Calling Sequence: CALL XSORTF(KEY,LR,NR,IB)

Purpose: Sorts NR records in a core block IB of records having fixed length LR in ascending binary sequence. The records are to be sorted in terms of the word within the record whose index within the record is specified by KEY.

3.9.2.44 XSTORE

Subprogram Type: Subroutine

Calling Sequence: CALL XSTORE(NAME,VALUE)

Purpose: Stores a value into the Executive System Parameter specified by NAME (A8). The type of the value is expected to correspond to type defined for the parameter. Valid input names and values follow:

Valid Name	Residence	Type	Range of Values
NERR	/XCVT/	LOGICAL	.TRUE. only

Description: Executive System parameter for error encountered while executing a control statement. Functional module sets NERR to .TRUE. to indicate an abnormal termination upon return to Executive Manager.

3.9.2.45 XTBDMP

Subprogram Type: Subroutine

Calling Sequence: CALL XTBDMP(ITBL,ITYP)

Purpose: XTBDMP dumps the system table in array ITBL having the table type specified by ITYP. Valid system table types are 1, 2, or 3.

3.9.2.46 XTRACE

Subprogram Type: Subroutine

Calling Sequence: CALL XTRACE(LIMIT)

Purpose: XTRACE provides a subroutine traceback capability which prints the names of the called and calling subroutines and the lines from which the called routine was called. The input variable limit indicates the name (A6) of the subroutine to which to trace or the integer number of levels to traceback. If LIMIT is zero or negative, a trace to the primary overlay level is done.

3.9.2.47 XT1AL

Subprogram Type: Integer Function

Calling Sequence: XT1AL(LOC)

Purpose: Calculates the current allocated length of a system Table Type 1 given LOC, the first word of the table preface.

3.9.2.48 XT1FV

Subprogram Type: Subroutine

Calling Sequence: CALL XT1FV(ITBL,KEYVAL,KEYLOC,ICONT,IPOS)

Purpose: Searches a Type 1 System Table or Directory, ITABL, for an entry having the specified value, KEYVAL, in a specific position, KEYLOC, within the entry.

3.9.2.49 XT2AL

Subprogram Type: Integer Function

Calling Sequence: XT2AL(LOC)

Purpose: Calculates the current allocated length of a System Table Type 2 given LOC, the first word of the table preface.

3.9.2.50 XT3FL

Subprogram Type: Subroutine

Calling Sequence: CALL XT3FL(IT,IC,IP)

Purpose: Locates the position, IP, of the last entry in a Type 3 Table given IT, the array containing the Type 3 Table, and IC, the position of the character pointer in the table preface.

3.9.2.51 XT3FV

Subprogram Type: Subroutine

Calling Sequence: CALL XT3FV(ITBL,ICHAIN,KEYVAL,KEYLOC,ICONT,IPOS)

Purpose: Searches a Type 3 Table or Directory for an entry in chain, ICHAIN, having the value, KEYVAL, in the specified location, KEYLOC, within the entry. Valid values for ICHAIN are NT3USD, used entry chain, and NT3ØTR, other entry chain.

3.9.2.52 XT3IF

Subprogram Type: Subroutine

Calling Sequence: CALL XT3IF(IT)

Purpose: Initializes new entries in the free chain of a Type 3 Table given IT, the array containing the Table.

3.9.2.53 XT3LK

Subprogram Type: Subroutine

Calling Sequence: CALL XT3LK(IT,IC,IP)

Purpose: Links an entry into a Type 3 Table Chain.

3.9.2.54 XUNPK

Subprogram Type: Subroutine

Calling Sequence: CALL XUNPK(IN,NC,IØUT)

Purpose: Unpacks NC characters from the word IN (A8) into the array IØUT (A1). NC must have a value greater than or equal to zero and less than or equal to the number of characters per word.

3.9.2.55 XUNPKM

Subprogram Type: Subroutine

Calling Sequence: CALL XUNPKM(IN,NC,IØUT)

Purpose: Unpacks NC characters from the word array IN (A8) into the string array IØUT (A1). NC must have a value greater than or equal to zero.

3.9.2.56 XUNPKT

Subprogram Type: Subroutine

Calling Sequence: CALL XUNPKT(IN,NC,IØUT,MAXØUT,LUØUT,ØVFL)

Purpose: Unpacks NC characters from the word array IN (A8) into the string array IØUT (A1) and truncates an overflow. NC must have a value greater than or equal to zero.

3.9.2.57 XVNAME

Subprogram Type: Logical Function

Calling Sequence: XVNAME(NAME)

Purpose: Determines if input argument NAME (A8) is a valid name.

3.9.2.58 XZFILL

Subprogram Type: Subroutine

Calling Sequence: CALL XZFILL(NAME)

Purpose: Removes the trailing blanks in argument NAME and replaces these blanks with zeroes. In the event there are blanks preceeding the left most character in the name or blanks embedded in the name, these blanks will remain unchanged.

3.9.3 Auxiliary Modules

A Utility error processer may be called by any one of the General Utilities if an error condition is encountered during its execution. Currently there is only need for a fatal error processor.

3.9.3.1 Utility Fatal Error Message Writer (XUFMSG)

Subroutine XUFMSG (NM,CNAME,VAR1,VAR2) processes the fatal utility errors by printing the error message indicated by NUM and aborting through a call to XEXIT.

3.9.3.2 System Tables Utility Error Message Writer (XTBERR)

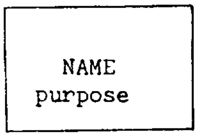
Subroutine XTBERR (NAME, IERR, IARG, IVAL, ITBL, IPL) processes the error messages for some of the utilities which manipulate system tables. NAME is the calling subprogram and IERR is the error number. If IERR is negative, the error is fatal and if positive non-fatal. IARG and IVAL contain informative values pertinent to the error encountered. ITBL and IPL permit the dumping of a table preface where ITBL is an array containing the table preface to be dumped and IPL is the preface length. If IPL = 0, no table will be dumped.

3.9.4 Hierarchy Charts

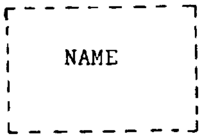
A hierarchy chart is a graphical representation of the logical relationships between modules. Figures 1-12 are the hierarchy charts for the General Utility modules and the auxiliary modules.

All General Utility modules appear at least once as a block entity in the hierarchy charts. Detail is to the lowest level module except when the called module is a service module (another utility, or a DSM or DBM module), the auxiliary module XUFMSG, or a subprogram provided by one of the CDC operating system libraries. However, these modules are listed in the hierarchy chart figures.

Symbols and headings used in the hierarchy charts are given below:



NAME - module name
purpose - brief description



represents logical module not existing as entity. It is used for logical groupings.

indicates lower level module is called by higher level module.

GENERAL UTILITIES

----------	implies logical grouping with no direct relationship.
*	in upper right corner of module block indicates module is expanded as a separate hierarchy.
ANOPP MODULE CALLED:	a list of DBM, DSM, and General Utility modules called by the modules in this figure.
CDC System Library Subprograms Called:	a list of subprograms called by the modules in this figure and which are not part of ANOPP system libraries.

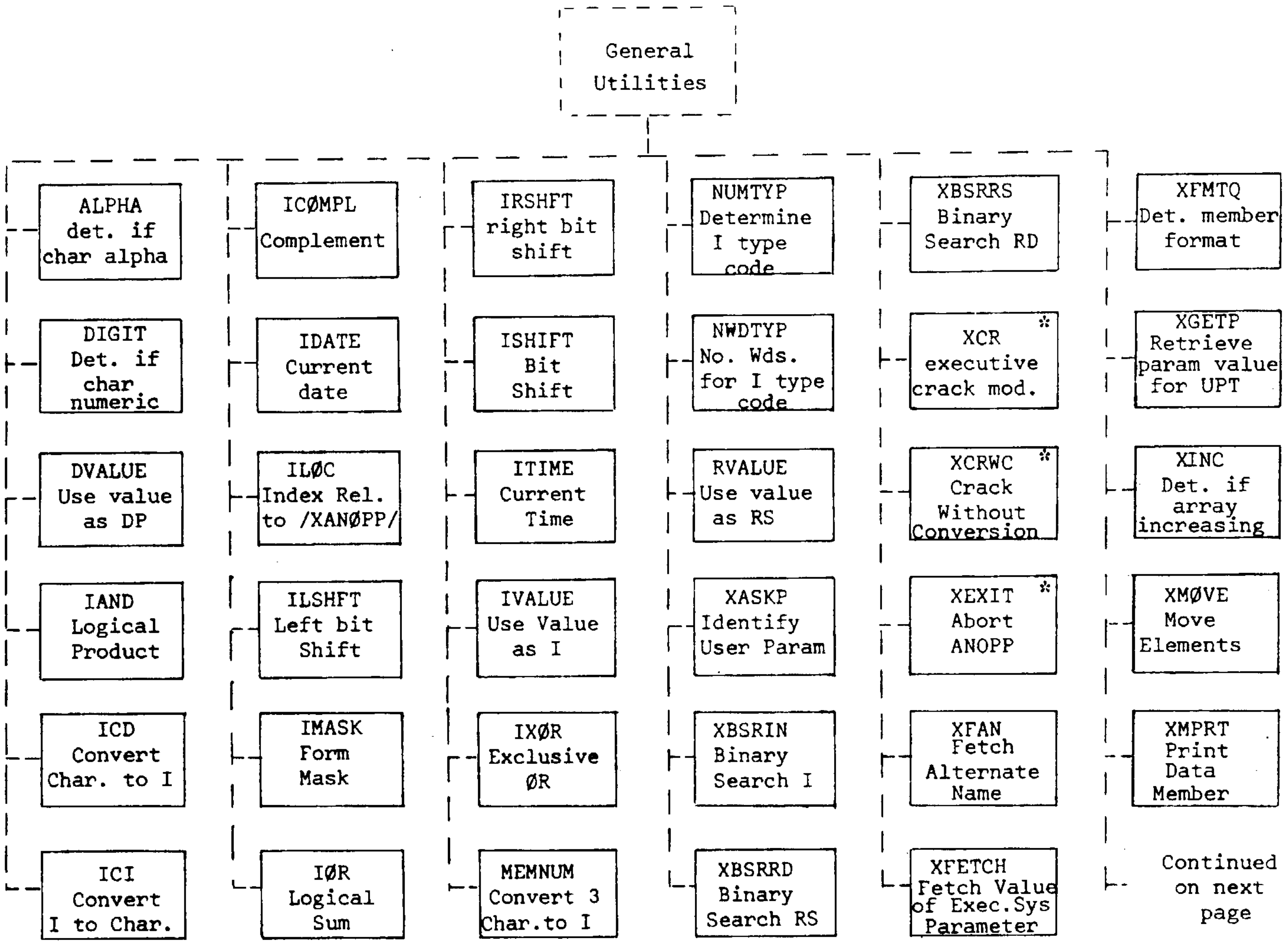
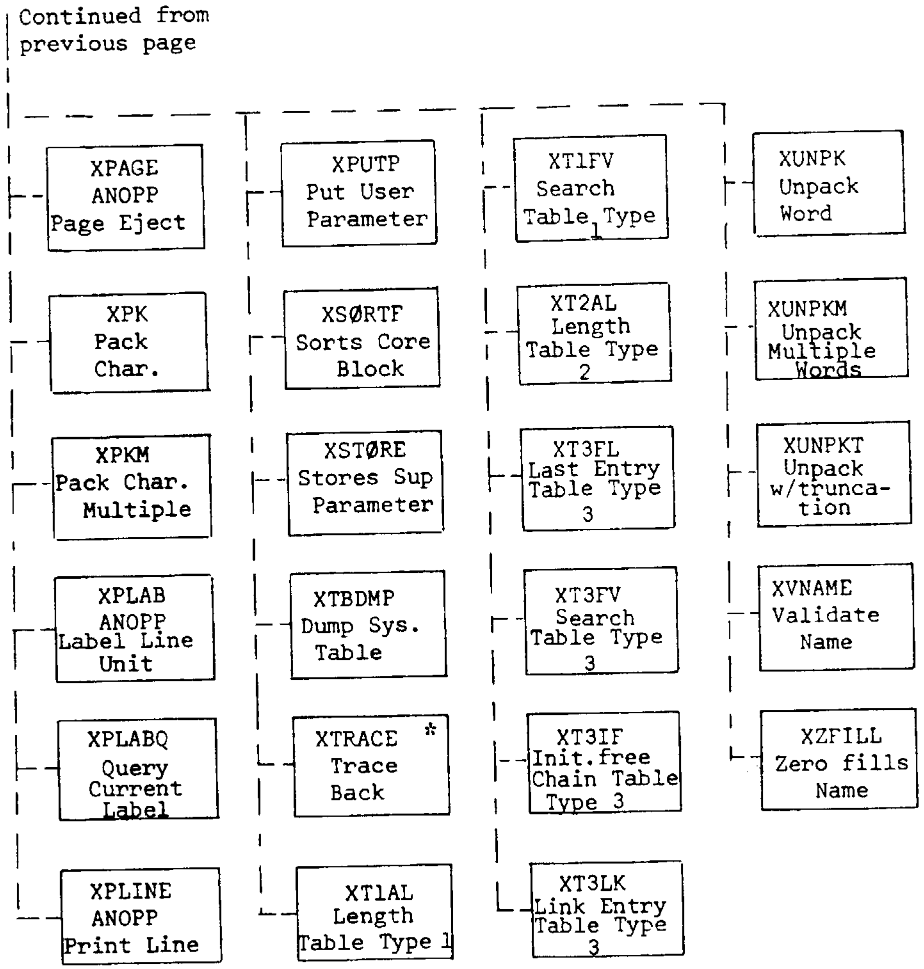


Figure 1. General Utility Hierarchies


Continued from previous page
XPAGE ANOPP Page Eject
XPK Pack Char.
XPKM Pack Char. Multiple
XPLAB ANOPP Label Line Unit
XPLABQ Query Current Label
XPLINE ANOPP Print Line
XPUTP Put User Parameter
XSØRTF Sorts Core Block
XSTØRE Stores Sup Parameter
XTBDMP Dump Sys. Table
XTRACE Trace Back *
XT1AL Length Table Type 1
XT1FV Search Table Type 1
XT2AL Length Table Type 2
XT3FL Last Entry Table Type 3
XT3FV Search Table Type 3
XT3IF Init. free Chain Table Type 3
XT3LK Link Entry Table Type 3
XUNPK Unpack Word
XUNPKM Unpack Multiple Words
XUNPKT Unpack w/truncation
XVNAME Validate Name
XZFILL Zero fills Name

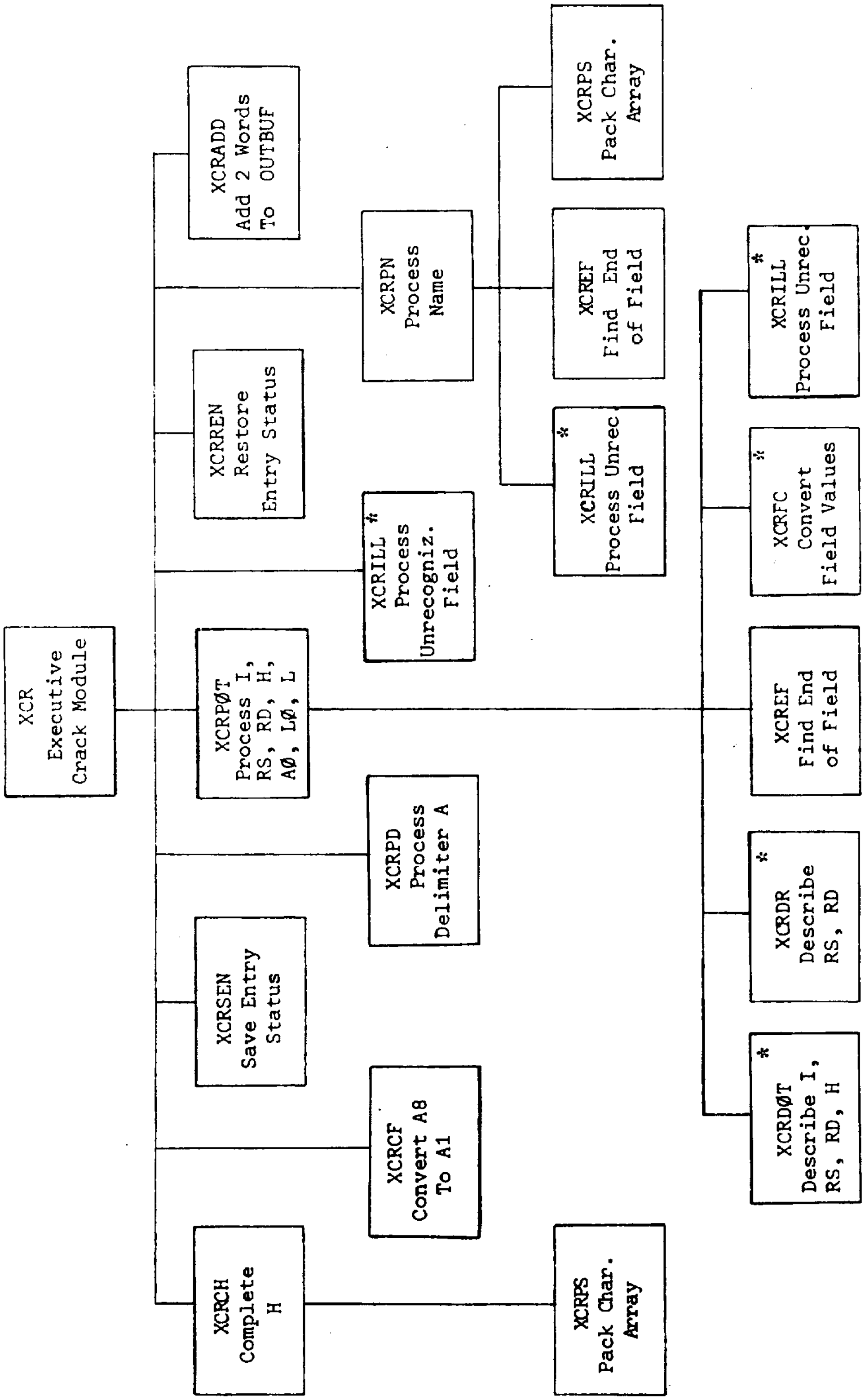
ANOPP Modules Called:

DSMX, MMCLØS, MMEDNM,
MMGETR, MMGETW, MMØPRD,
TMERR, XFMTQ, XTBERR,
XUFMSG, XXFMSG, XXNMSG

CDC System Library Subprograms Called:

AND, CØMPL, DATE,
IABS, LØCF, MASK,
MØD, ØR, SHIFT,
TIME, XØR

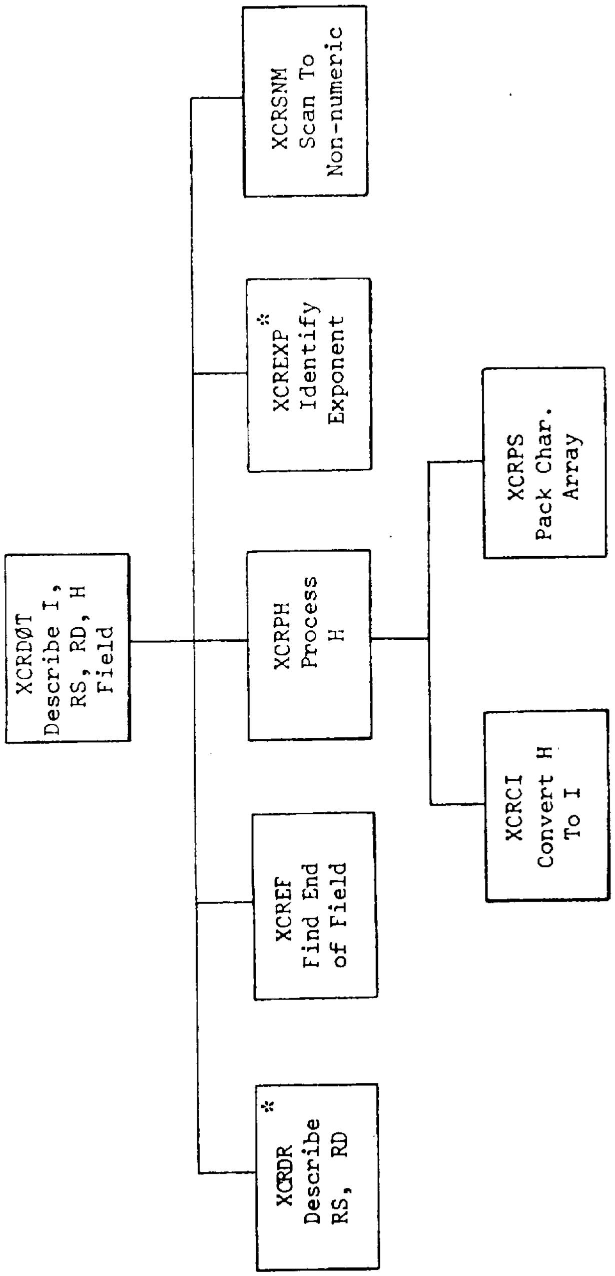
Figure 1. General Utility Hierarchies (Continued)



ANOPP Modules Called:

ALPHA, DIGIT, DSMF, DSMG,
XPF, XUFMSG, XUNPK

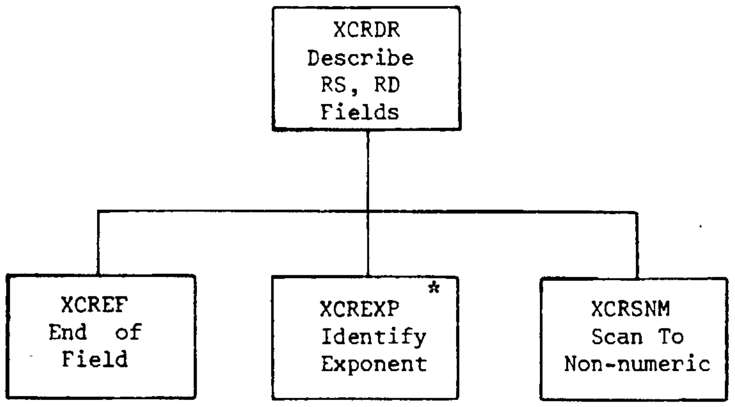
Figure 2. XCR Hierarchy Chart



ANOPP Modules Called:

DIGIT, ICD, XPK

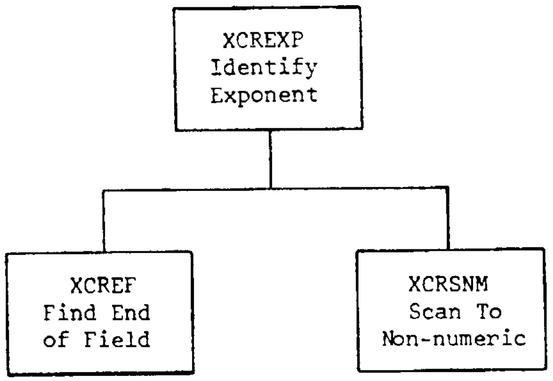
Figure 3. XCRDØT Hierarchy Chart



ANOPP Modules Called:

DIGIT

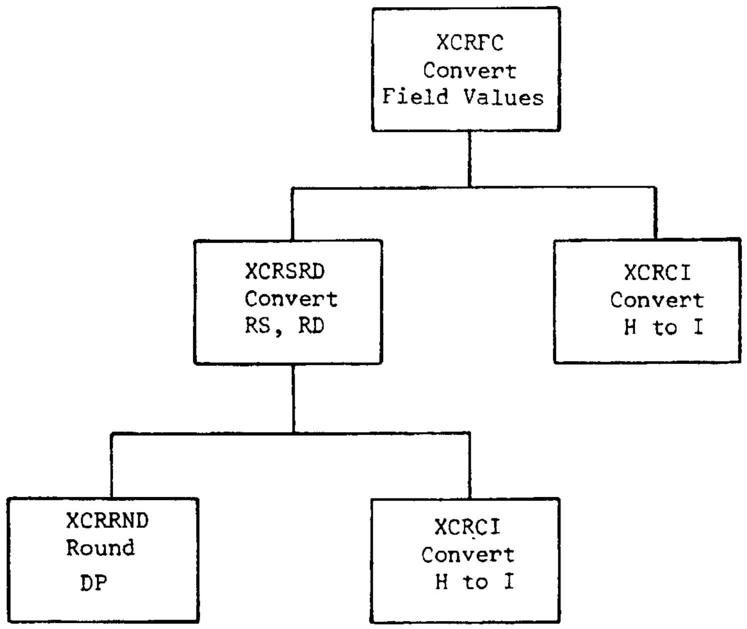
Figure 4. XCRDR Hierarchy Chart



ANOPP Modules Called:

DIGIT

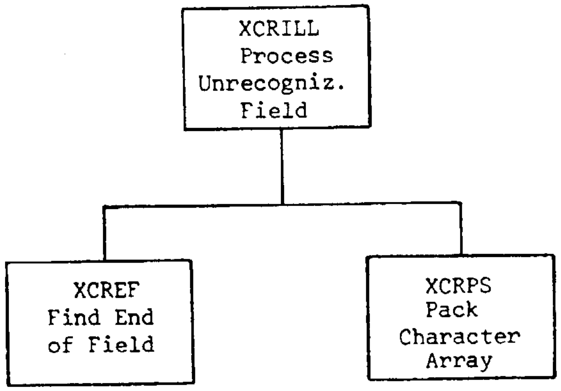
Figure 5. XCREXP Hierarchy Chart



ANOPP Modules Called:

DVALUE, ICD, ILSHIFT,
IRSHFT, RVALUE, XPK

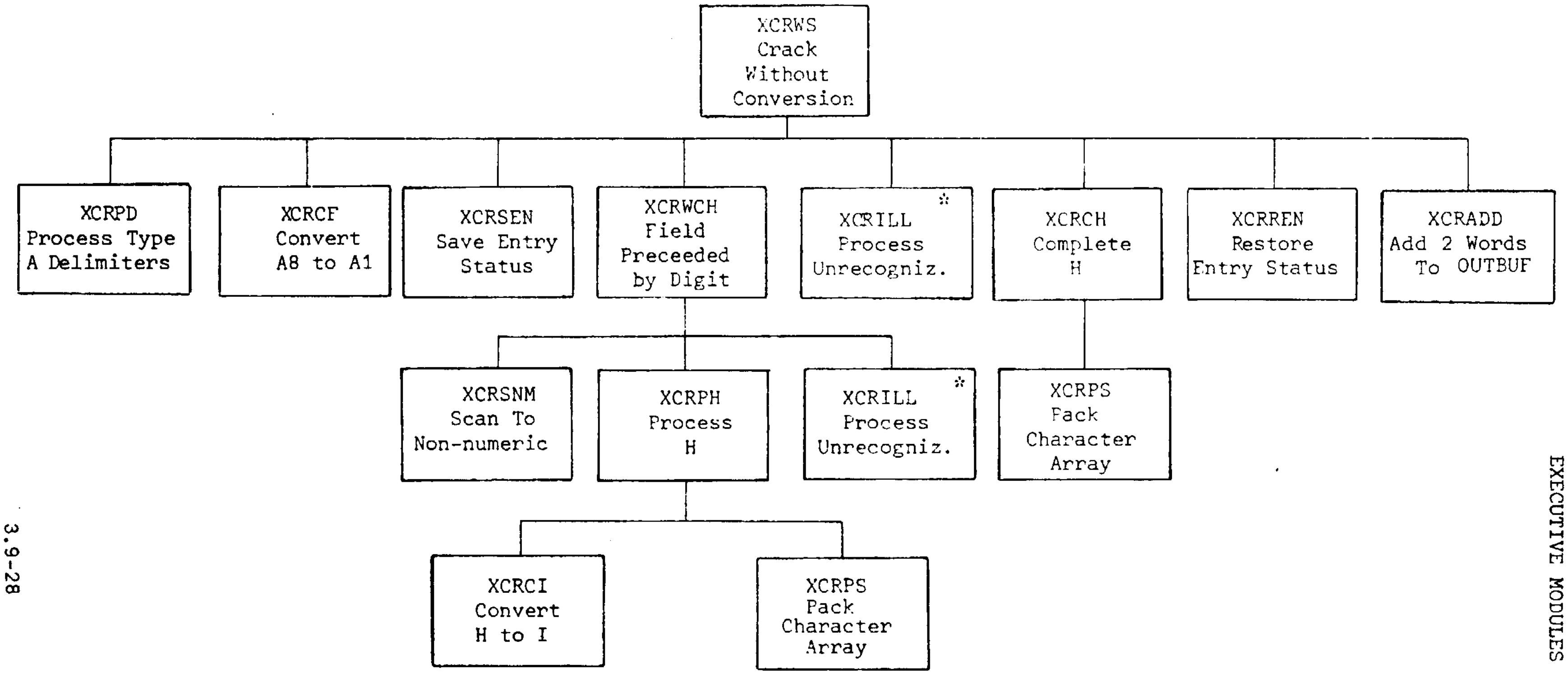
Figure 6. XCRFC Hierarchy Chart



ANOPP Modules Called:

XPK

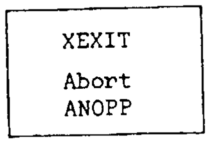
Figure 7. XCRILL Hierarchy Chart



ANOPP Modules Called:

DIGIT, DSMF, DSMS, ICD,
XPK, XUFMSG, XUNPK

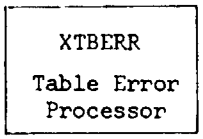
Figure 8. XCRWC Hierarchy Chart



ANOPP Modules Called:

XFETCH, XPLAB, XPLINE,
XTRACE

Figure 9. XEXIT Hierarchy Chart



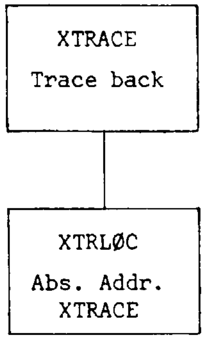
ANOPP Modules Called:

XEXIT, XPLINE

CDC System Library Subprograms Called:

IABS

Figure 10. XTBERR Hierarchy Chart



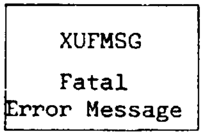
ANOPP Modules Called:

XPLINE

CDC System Library Subprograms Called:

```
AND,     LØCF,     MASK,
ØR,      SHIFT
```

Figure 11.   XTRACE Hierarchy Chart



ANOPP Modules Called:

RVALUE, XEXIT, XFETCH, XPLINE

Figure 12. XUFMSG Hierarchy Chart

MACHINE DEPENDENT INFORMATION

4.1 OVERVIEW

The following subsections describe the procedures for installation and execution of ANOPP and of ANOPP Functional Modules. These procedures depend upon the host computer operating system, and the loader used to accomplish the loading of multiple overlay segments.

While the examples given in this section have been tested and can be used in "cookbook" fashion, many variations of these basic examples are available to anyone with a working knowledge of the specific file oriented operating system and loader being used. For such knowledge, the reader is referred to the references at the end of each subsection. The examples given are specific solution to the general problem of getting the right information on the right file, in the right place, at the right time.

4.2 CDC CYBER NOS WITH THE NASTRAN LINKAGE EDITOR

Five files are of recurring interest during installation and execution procedures. They are a source file, an object file, an executable file, a loader directives file and a library file. While these files may be assigned any number of valid names, they have been assigned mnemonic names such that their use will be more apparent in the procedures described. The names are:

ANØPL ANOPP source file in CDC UPDATE Program Library format.

ANØPB object file of relocatable load modules, output from FORTRAN compilation and input to Linkage Editor.

ANØPP executable load file, output from Linkage Editor.

SUBSYS Linkage Editor segmentation directives file in CDC UPDATE Program Library format.

LINKLIB load library used by the Linkage Editor to satisfy external references.

4.2.1 Installation Procedures

There are four basic installation procedures. In increasing level of complexity, they are:

1. generate an executable file
2. modify an existing module
3. install a dummy functional module
4. install a new functional module

These procedures involve the five important files, ANØPL, ANØPB, ANØPP, SUBSYS, and LINKLIB. At each step the user has the choice of making either temporary or permanent changes to these files. These options will be discussed in each procedure without giving an example of every possible combination.

4.2.1.1 Generate An Executable File

The executable file, ANØPP, is output in random access mode from Linkage Editor processing. The Linkage Editor requires binary load modules, segmentation directives, and a LINKLIB. The sample job in Figure 1 illustrates the binary load modules coming from ANØPB and the segmentation directives from CØMPILE which was output from a CDC UPDATE of SUBSYS. ANØPB is output from compilation of source code that came from an UPDATE of ANØPL. In this example, the compilation listing and directive listing are printed for future reference, and the binary load modules and executable file are permanently saved.

4.2.1.2 Modify An Existing Module

If a module in the permanently resident segment, LINKO, is to be modified, then a full Linkage Editor run must be made to regenerate every link on the executable ANØPP file. Otherwise, it is possible to do a partial Linkage Editor run to regenerate only those links containing the modified module or modules. The full case uses all of the directives from the SUBSYS file as in generating an executable

```
JOB.
ACCOUNT.
CHARGE.
ATTACH,OLDPL=ANOPL/NA.
UPDATE,F,L=A1,C=SOURCE,I=0.
DEFINE,ANOPB.
FTN,I=SOURCE,B=ANOPB.
ATTACH,SUBSYS/NA.
UPDATE,F,L=1,P=SUBSYS,I=0.
COPYSBF,COMPILE,OUTPUT.
ATTACH,LINKEDT,LINKLIB/UN=837250C,NA.
DEFINE,ANOPP.
REWIND,COMPILE.
RFL,200000.
LINKEDT,COMPILE.
RFL,150000.
REDUCE,-.
MAP,OFF.
LDSET,LIB=FORTRAN/SYSIO.
ANOPP,ATTACH
7/8/9 EOR **************************** EOR ****************************** EOR 7/8/9
STARTCS $
ENDCS $
6/7/8/9 EOF ************************** EOF **************************** EOF 6/7/8/9
-
```

Figure 1. Generate System From Source Code

file. In the partial case a copy of an old executable file is declared as an INFILE on the LINKEDIT directive and only those directives pertaining to the affected links are selected from SUBSYS. In either case, the modified source is obtained from an UPDATE of the ANØPL file. The modified source is compiled and the corresponding modified binary load modules are placed on an LGØ file. The appropriate directives are selected from SUBSYS and input to the Linkage Editor along with LGØ, ANØPB, and LINKLIB. The Linkage Editor will search for load modules on LGØ and ANØPB and will give preference to modules named on LGØ if duplicates occur. Figure 2 gives an example of a full Linkage Editor run with the executable ANØPP file produced in sequential mode. Figure 3 gives an example of a partial Linkage Editor run with the executable ANØPP file produced in random access mode. In Figure 2, the direct access permanent file, ANØPP, will be written in sequential form by the Linkage Editor and must be executed subsequently with an ANØPP. control card. In Figure 3, the direct access permanent file, ANØPP, was written in random access format in a previous Linkage Editor run and is copied to a local file, MYFILE, that is used as both INFILE and ØUTFILE in random access format during this Linkage Editor run. The file, MYFILE, is executed subsequently with a MYFILE.ATTACH control card.

4.2.1.3 Temporarily Install A Dummy Functional Module

The standard executable version of ANOPP has provided for five dummy functional modules named FM1 through FM5 to reside in links 5 through 9 respectively. For purposes of this discussion it will be assumed that the user has sufficient knowledge of the ANOPP executive system and its interfaces to have written the source code for a functional module and now wishes to install and execute a test of the module. To do this, the user must first understand that the ANOPP control statement EXECUTE FM1 will really cause the module in LINK5 to be loaded and executed. Likewise for FM2/LINK6, FM3/LINK7, etc. Thus, the user must execute FM5 to test a dummy module in LINK9, but the names of the routines in LINK9 need not be FM5. The names of the routines in LINK9 are determined by the INCLUDE directives for that link and must have been found on LGØ or ANØPB. The following two examples illustrate these

```
JOB•
ACCOUNT•
CHARGE•
ATTACH•OLDPL=ANOPL/NA•
UPDATE•Q•L=1•C=CHANGE•
FTN•I=CHANGE•
ATTACH•SUBSYS/NA•
UPDATE•Q•L=1•P=SUBSYS•
ATTACH•ANOPP/M=W•NA•
ATTACH•ANOPB/NA•
ATTACH•LINKEDT•LINKLIB/UN=837250C•NA•
RFL•200000•
LINKEDT•COMPILE•
RFL•150000•
REDUCE•-•
MAP•OFF•
LDSET•LIB=FORTRAN/SYSIO•
ANOPP•
7/8/9 EOR *************************** EOR ***************************** EOR 7/8/9
*ID CHANGE
*/ UPDATE CHANGES AND MODIFICATIONS• AS APPROPRIATE TO SOURCE CODE
*C APPROPRIATE DECK(S)
7/8/9 EOR *************************** EOR ***************************** EOR 7/8/9
*ID CHANGE
*D LINKEDIT•2
LINKEDIT LET•OUTFILE=ANOPP(T)•PARAM(6)=15000
*C LINKEDIT•ENDLINKS
7/8/9 EOR *************************** EOR ***************************** EOR 7/8/9
STARTCS $
ENDCS $
6/7/8/9 EOF ************************* EOF *************************** EOF 6/7/8/9
-
```

Figure 2. Update Source Code and Perform Full Linkage Editor Run

```
JOB.
ACCOUNT.
CHARGE.
ATTACH.OLDPL=ANOPL/NA.
UPDATE.Q.L=1.C=CHANGE.
FTN.I=CHANGE.
ATTACH.SUBSYS/NA.
UPDATE.Q.L=1.P=SUBSYS.
ATTACH.ANOPP/NA.
COPYEI.ANOPP.MYFILE.
ATTACH.ANOPB/NA.
ATTACH.LINKEDT.LINKLIB/UN=837250C.NA.
RFL.150000.
LINKEDT.COMPILE.
REDUCE.-.
MAP.OFF.
LDSET.LIB=FORTRAN/SYSIO.
MYFILE.ATTACH
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
*ID CHANGE
*/ UPDATE CHANGES AND MODIFICATIONS. AS APPROPRIATE
*C APPROPRIATE DECK(S) (NOT IN LINK 0)
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
*ID CHANGE
*D LINKEDIT.2
LINKEDIT LET.INFILE=MYFILE(C).OUTFILE=MYFILE(C).PARAM(6)=15000
*C LINKEDIT.ENDLINKS
*C APPROPRIATE LINK (OTHER THAN LINK 0)
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
STARTCS $
ENDCS $
6/7/8/9 EOF ************************* EOF ************************** EOF 6/7/8/9
-
```

Figure 3. Update Source Code and Perform Partial Linkage Editor Run

concepts. In the first, a dummy module consisting of the single subroutine FM2 has been installed in LINK6, and secondly a dummy module consisting of three subroutines has been installed in LINK8.

In Figure 4, the dummy module source code consisting of the single subroutine FM2 is compiled and the relocatable load module FM2 is placed on the file LGØ. Then a partial Linkage Editor run is performed using the set of directives for LINK6 called from SUBSYS. These directives already have an INCLUDE statement for the dummy routine FM2. Then the new executable version of ANOPP on the local random file MYFILE is executed and the dummy functional module is tested via the EXECUTE FM2 control statement.

In Figure 5, the dummy module source code consisting of the three routines MYMØD, MYMØDA, and MYMØDB is compiled and the three relocatable load modules are placed on the file LGØ. The Linkage Editor directives for LINK8 are supplied from INPUT to reflect the overlay structure desired for the three routines of the dummy module to be tested. A partial Linkage Editor run is made to construct a new version of the ANOPP executable code on a local file, MYFILE. This version contains the new dummy routines in LINK8. The dummy module can be tested by placing an EXECUTE FM4 card in the ANOPP control statement set and executing an ANOPP run via the MYFILE.ATTACH control card.

4.2.1.4 Permanently Install A New Functional Module

Permanent installation of a new module requires some minor modificiations to part of the ANOPP executive management system. The primary concern is to match the new module name with the new link in which it resides. This is accomplished via matching entries in the two arrays FMN and IFMN in the subprogram XRTSEX. The length and contents of these arrays are controlled by INTEGER and DATA statements in the same subprogram XRTSEX. The number of active entries in each array is controlled by the variable NFM in the common block /XCS/ and is set by a DATA statement in the subprogram XCSBD. Array FMN contains the names of existing functional modules, and array IFMN contains the corresponding link numbers. At present, dummy modules FM1 through FM5 in links 5 through 9 are permanently installed.

```
JOB.
ACCOUNT.
CHARGE.
FTN.
ATTACH,ANOPP/NA.
COPYEI,ANOPP,MYFILE.
ATTACH,OLDPL=SUBSYS/NA.
UPDATE,Q,L=1.
ATTACH,ANOPB/NA.
ATTACH,LINKEDT,LINKLIB/UN=837250C,NA.
RFL,150000.
LINKEDT,COMPILE.
MAP,OFF.
REDUCE,-.
LDSET,LIB=FORTRAN/SYSIO.
MYFILE.ATTACH
7/8/9 EOR **************************** EOR ***************************** EOR 7/8/9
*
*      SOURCE DECK FOR DUMMY FUNCTIONAL MODULE FM2
*
7/8/9 EOR **************************** EOR ***************************** EOR 7/8/9
*ID DUMMY
*D LINKEDIT.2
LINKEDIT LET,INFILE=MYFILE(C),OUTFILE=MYFILE(R)PARAM(6)=15000
*C LINKEDIT,ENDLINKS
*C LINK6
7/8/9 EOR **************************** EOR ***************************** EOR 7/8/9
STARTCS $
$     ANOPP CONTROL STATEMENTS, AS APPROPRIATE
EXECUTE FM2 $
$     ANOPP CONTROL STATEMENTS, AS APPROPRIATE
ENDCS $
6/7/8/9 EOF ************************** EOF *************************** EOF 6/7/8/9
-
```

Figure 4. Sample Run To Execute Dummy Functional Module FM2

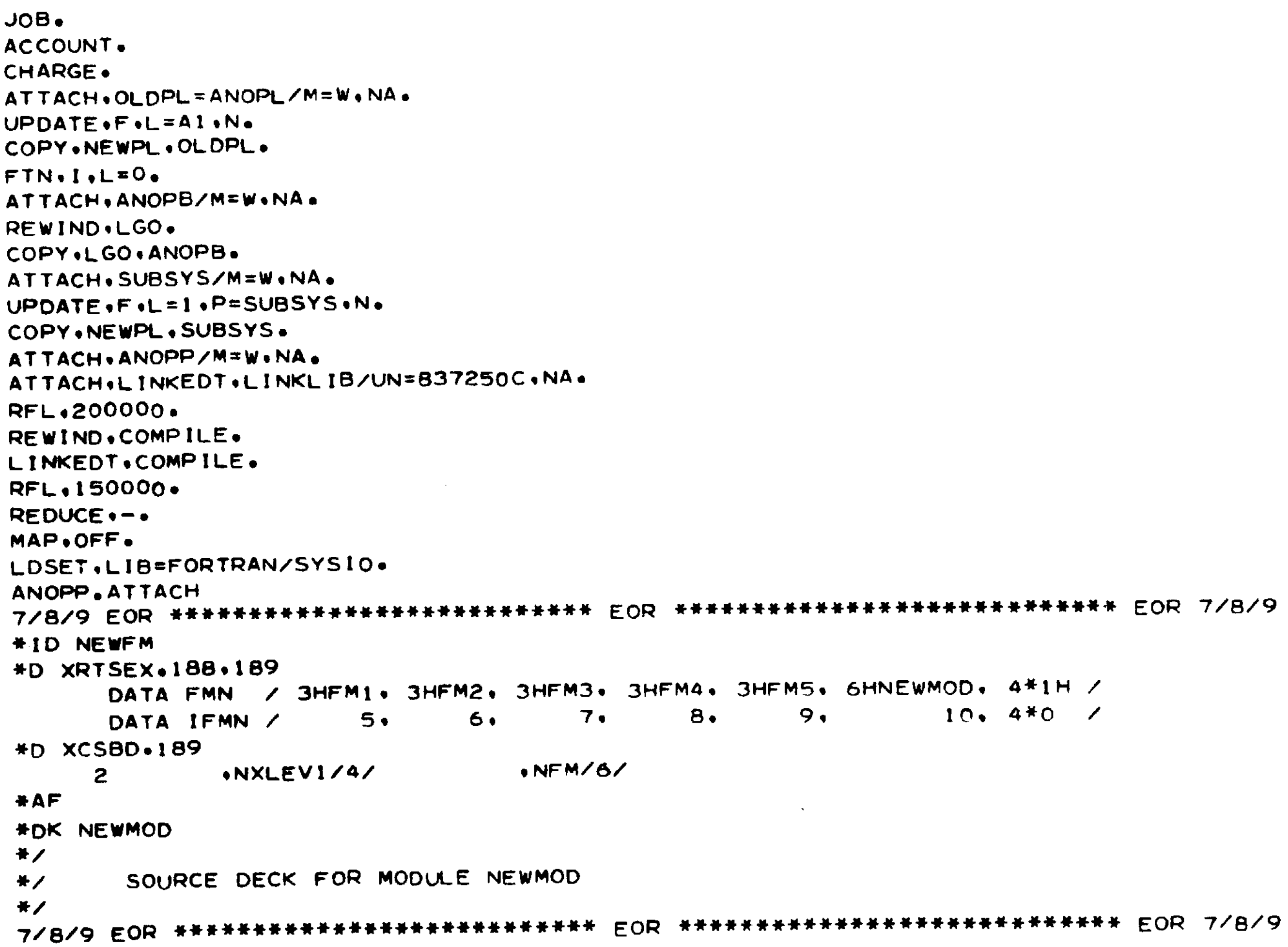
```
JOB.
ACCOUNT.
CHARGE.
FTN.
ATTACH.ANOPP/NA.
COPYEI.ANOPP.MYFILE.
ATTACH.ANOPB/NA.
ATTACH.LINKEDT.LINKLIB/UN=837250C.NA.
RFL.150000.
LINKEDT.
MAP.OFF.
REDUCE.-.
LDSET.LIB=FORTRAN/SYSIO.
MYFILE.ATTACH
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
*
*     SOURCE DECKS FOR DUMMY FUNCTIONAL MODULE
*
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
LINKEDIT LET.INFILE=MYFILE(R).OUTFILE=MYFILE(C).PARAM(6)=15000
LIBRARY ANOPB=LGO.ANOPB
LINK 8
ENTRY MYMOD
INCLUDE ANOPB (MYMOD     )
OVERLAY ONE
INCLUDE ANOPB (MYMODA    )
OVERLAY ONE
INCLUDE ANOPB (MYMODB    )
END
ENDLINKS
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
STARTCS $
$     ANOPP CONTROL STATEMENTS. AS APPROPRIATE
EXECUTE FM4 $
$     ANOPP CONTROL STATEMENTS. AS APPROPRIATE
ENDCS $
6/7/8/9 EOF ************************* EOF ************************** EOF 6/7/8/9
-
```

Figure 5. Sample Run To Temporarily Install Dummy Functional Module

MACHINE DEPENDENT INFORMATION

In Figure 6, an example of adding a new module with the symbolic name NEWMØD is given. The module will be placed in LINK10. To begin with, the source code for NEWMØD and the required changes to the executive management modules is obtained by an UPDATE of the source file ANØPL. In this example, the NFM variable is increased to 6 and the name NEWMØD is placed in the array FMN while the link number 10 is added to the array IFMN. After compilation, the binary load modules are placed on the file LGØ. A full Linkage Editor run must be performed since a subprogram in LINKO is being changed. Thus, all of the directives from the SUBSYS file plus the new LINK10 directives must be used. The new version of ANOPP is executed by an ANØPP.ATTACH card and the new module is tested with an EXECUTE NEWMØD control statement.

In this example, it is assumed that the new module was previously tested thoroughly as a dummy module and that the installation is intended to be permanent. Therefore new versions have been created or rewritten for the source file, ANØPL; the binary file, ANØPB; the directives file, SUBSYS; and the executable file, ANØPP.



```
JOB.
ACCOUNT.
CHARGE.
ATTACH,OLDPL=ANOPL/M=W,NA.
UPDATE,F,L=A1,N.
COPY,NEWPL,OLDPL.
FTN,I,L=0.
ATTACH,ANOPB/M=W,NA.
REWIND,LGO.
COPY,LGO,ANOPB.
ATTACH,SUBSYS/M=W,NA.
UPDATE,F,L=1,P=SUBSYS,N.
COPY,NEWPL,SUBSYS.
ATTACH,ANOPP/M=W,NA.
ATTACH,LINKEDT,LINKLIB/UN=837250C,NA.
RFL,200000.
REWIND,COMPILE.
LINKEDT,COMPILE.
RFL,150000.
REDUCE,-.
MAP,OFF.
LDSET,LIB=FORTRAN/SYSIO.
ANOPP.ATTACH
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
*ID NEWFM
*D XRTSEX.188,189
      DATA FMN  / 3HFM1, 3HFM2, 3HFM3, 3HFM4, 3HFM5, 6HNEWMOD, 4*1H /
      DATA IFMN /     5,     6,     7,     8,     9,        10, 4*0  /
*D XCSBD.189
     2        ,NXLEV1/4/          ,NFM/6/
*AF
*DK NEWMOD
*/
*/      SOURCE DECK FOR MODULE NEWMOD
*/
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
```

(Continued on next page)

Figure 6. Sample Run To Permanently Install A New Functional Module

(continued from previous page)

```
*AF •LINK9
*DK LINK10
LINK 10
ENTRY NEWMOD
INCLUDE ANOPB (NEWMOD    )
END
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
STARTCS $

$    ANOPP CONTROL STATEMENTS• AS APPROPRIATE
EXECUTE NEWMOD $
$    ANOPP CONTROL STATEMENTS• AS APPROPRIATE
ENDCS $
6/7/8/9 EOF ************************* EOF ************************** EOF 6/7/8/9
-
```

Figure 6. Sample Run To Permanently Install A New Functional Module (continued)

4.2.2 Execution Procedure

Execution procedures involve both the external host computer operating system environment and the internal ANOPP executive and data base management system. The external system is concerned with initiation of program loading and access to system files. The internal system is concerned with interface and access to and between external files and internal data base structures. Some operations must occur in pairs or combinations while other procedures or control cards act independently.

4.2.2.1 Program Loading

Loading procedures for the executable program vary depending upon the random or sequential mode of the file to be loaded. The executable file is a double file containing a bootstrap program and executable program separated by a file mark. The control card that loads the executable file actually loads the bootstrap loader program that then accomplishes loading of the executable program. The actions taken by this bootstrap loader are determined by the form of the control card. If the bootstrap loader program and executable program reside on an executable file named ANØPP, then the loading options are:

1.
```
LDSET,LIB=FØRTRAN/SYSIØ.
ANØPP.ATTACH
```

   This form is used by the bootstrap loader to execute an executable file that exists in random format. A random format is produced by the Linkage Editor as an ØUTFILE with R or C status. It can also be produced from a sequential file by the bootstrap loader under the CATLØG option (see 2. below).

2.
```
LDSET,LIB=FORTRAN/SYSIØ.
ANØPP.CATLØG(XXX)
LDSET,LIB=FORTRAN/SYSIØ.
XXX.ATTACH
```

   These two control cards must appear in a pair and the name xxx may be any valid file name, but must be the same name on both cards. The CATLØG command instructs the bootstrap loader to transform a sequential executable file ANØPP into a random executable file xxx. The random file xxx is then executed with an ATTACH command similar to 1 above. The file ANØPP must be sequential and could only have been produced by the Linkage Editor as an ØUTFILE with S or T status.

3.
```
LDSET,LIB=FORTRAN/SYSIØ.
ANØPP.
```

This form may be used to execute a sequential executable file output from the Linkage Editor. In this case, the bootstrap loader internally generates the equivalent of 2 above in the form ANØPP.CATLØG(SYSLMØD) followed by SYSLMØD. ATTACH. This form may not be used with random executable files.

The NØS operating system and FORTRAN extended language together provide the option of changing the names of program files at execution time. This is accomplished via an order dependent substitution of file names on the control card that initiates program loading. For ANØPP, the two main program files are INPUT and ØUTPUT as declared in the main program XM. Alternate file names can be substituted for them by altering the ANØOPP.ATTACH and ANØPP. execution sequences above. Substitutions cannot be made with the ANØPP.CATLØG card, but may be placed in the xxx.ATTACH command. The substitutions are made by including the alternate names, separated by commas, between the final P in ANØPP and the period(.). The list is order dependent, but may terminate early. However, leading commas must be used to space over leading files for which no substitution is desired. For example:

```
ANØPP,ALTIN,ALTØUT.ATTACH
ANØPP,,ALTØUT.
XXX,ALTIN.ATTACH
```

4.2.2.2 Data Interfaces

Certain correspondences must be maintained between internal data units and external files during an ANOPP execution. Other correspondences can be noted from one execution to another. For example, an ATTACH of a data unit and external file must have been preceded by a CREATE of the internal data unit and a SAVE of the corresponding external file in another run. Most of these restrictions have been mentioned in descriptions of individual ANOPP control statements. Two examples of ANOPP executions that illustrate inter and intra job dependencies are presented.

The first job, in Figure 7, begins with attaching a direct access permanent file ANØPP which was previously output as a sequential executable file by the Linkage Editor. A sequential external file SEFN is accessed, a direct access permanent file EFN3 is at-

```
JOB.
ACCOUNT.
CHARGE.
ATTACH,ANOPP/NA.
GET,SEFN.
ATTACH,EFN3/M=W,NA.
DEFINE,ANOPX/CT=PU.
RFL,150000.
REDUCE,-.
MAP,OFF.
LDSET,LIB=FORTRAN/SYSIO.
ANOPP.CATLOG(ANOPX)
LDSET,LIB=FORTRAN/SYSIO.
ANOPX.ATTACH
SAVE,EFN2=UNIT2/CT=PU.
REPLACE,SEFN.
7/8/9 EOR ************************** EOR **************************** EOR 7/8/9
STARTCS $
CREATE UNIT1,UNIT2/EFN2/UNIT3/EFN3/$
DATA DM = MEMB1 $
EXECUTE NOISE UNIT=U MEMB=M $
END* $
UPDATE,NEWU=UNIT1,SOURCE=* $
-ADDR DATA(MEMB1) $
END* $
TABLE UNIT2(TBL1), 1, SOURCE=* $
INT = 0, 1
IND1 = RS, 3, 0, 1, 1.5, 2.0, 4.5
IND2 = 1, 2, 0, 1, 5, 10
DEP = 1, 3, 5, 7, 8, 9
END* $
DETACH,UNIT2$
ARCHIVE,UNIT3$
UNLOAD/SEFN/ $
PURGE UNIT3 $
ENDCS $
6/7/8/9 EOF ************************ EOF ************************** EOF 6/7/8/9
-
```

Figure 7. Sample Run To Create Member MEMB1

tached, and another direct access permanent file ANØPX is defined. The ANOPP program is loaded and executed via the CATLØG/ATTACH sequence. The sequential executable file ANØPP is transformed by the bootstrap loader into the random executable file ANØPX and then loaded and executed via the ANØPX.ATTACH control card. During execution, UNIT1 is created with a scratch external file name and UNIT2 is created with an external file name EFN2. UNIT3 is created and connected to the externally attached EFN3. Member MEMB1 is placed on unit DATA. Member MEMB1 is added from DATA to UNIT1 during the ANOPP UPDATE control statement processing. Member TBL1 is placed on UNIT2 during ANOPP TABLE control statement processing. UNIT2 is detached from Data Unit Directory but the external file name EFN2 remains open in the external system. UNIT3 is ARCHIVED. The remaining non-archived data unit in the Data Unit Directory, UNIT1, is unloaded to the sequential external file SEFN. The purge of UNIT3 causes the data unit name UNIT3 to be removed from the internal unit directory and the external file name EFN3 to be removed from the external system file table. Following ANOPP execution, the external file EFN2 is permanently saved as UNIT2 and the sequential file SEFN is replaced.

The second job, in Figure 8, begins by copying the ANOPP primary control statement set from INPUT to an alternate file ØTHERIN. Next the sequential external file from job 1, SEFN, is accessed and the external file equivalent to EFN2 in job 1 is accessed from the permanent file UNIT2. Then the previously created random executable file is obtained by an ATTACH of ANØPX and loading is initiated via a bootstrap loader ATTACH command with the alternate input file ØTHERIN substituted for INPUT. During ANOPP execution, the data unit UNIT2 is attached into the Data Unit Directory and connected with the existing external file UNIT2. UNIT1 is loaded from SEFN after which SEFN is removed from the Library File Directory and from the external system file table via the DRØP control statement. After all cracking and substituting is accomplished, the CALL control statement eventually leads to execution of the ANØPP control statement EXECUTE NØISE UNIT=UNIT2,MEMB=TBL1 $.

```
JOB.
ACCOUNT.
CHARGE.
COPY.INPUT.OTHERIN.
REWIND.OTHERIN.
GET.SEFN.
GET.UNIT2.
RFL.150000.
REDUCE.-.
MAP.OFF.
ATTACH.ANOPX/NA.
LDSET.LIB=FORTRAN/SYSIO.
ANOPX.OTHERIN.ATTACH
7/8/9 EOR *************************** EOR **************************** EOR 7/8/9
STARTCS $
ATTACH.UNIT2/UNIT2/$
LOAD/SEFN/ $
DROP /SEFN/ $
CALL UNIT1(MEMB1) U=UNIT2.M=TBL1 $
ENDCS $
6/7/8/9 EOF ************************* EOF ************************** EOF 6/7/8/9
-
```

Figure 8. Sample Run To Execute Previously Created Member MEMB1

4.2.3 CDC CYBER NOS DEPENDENCY

1. Coding Standards Violation;
2. Direct use of operating system capabilities; and
3. Interfacing the ANOPP Data Base Manager with the CYBER Record Manager.

The following paragraphs document what these dependencies are and where they are in ANOPP.

4.2.2.1 Standards Violations

1. Use of the intrinsic functions available through CDC CYBER FORTRAN EXTENDED.
2. Use of "no mode" arguments in subroutine and function calls.
3. Use of named block data subprograms.

4.2.3.2 Operating System Dependent Subprograms

a. XBSDFL - This CYBER COMPASS subprogram uses the MEM macro to determine the amount of central core memory available for the ANOPP run.

b. XPURGE - This CYBER COMPASS subprogram uses the UNLOAD macro to dispose of unwanted files.

c. MMCRMX - This FORTRAN subprogram is used by CYBER RECORD MANAGER if an error is found in accessing an ANOPP data unit.

d. XTRACE - This FORTRAN subprogram uses FORTRAN generated trace back structures.

The following subprograms use CDC CYBER Record Manager subprogram calls to perform input/output operations:

1. MMFEFB
2. MMGEFB
3. MMGET
4. MMMDMH
5. MMPUT
6. MMRMD
7. MMRMH
8. MMUHMD
9. XDR
10. XLDBGN
11. XLDEND
12. XLDFDM
13. XLDFDU
14. XLDLDM
15. XLDLDU
16. XUN
17. XUNBGN
18. XUNCPY
19. XUNEND
20. XUNLUH

4.2.3.3 I/Ø Interfaces

On CDC CYBER NOS, the ANOPP Data Base Manager makes use of the CYBER Record Manager (CRM) for all input and output. Data Units are written to word addressable files using unformatted records. ANOPP Library Files are generated using CRM Internal blocking (I) and control word (w) record format. On NOS, CRM has FORTRAN callable subprograms which are heavily used by ANOPP Member Manager.

APPENDIX A

GLOSSARY

Alternate Names - The set of names, established on the EXECUTE CS, which corresponds to a set of reference names. The set of alternate names is maintained in the Alternate Names Table and is available for retrieval by a functional or an executive system module during the execution of that functional module.

Cleanup Procedures - Functions performed upon completion of functional module to insure the integrity of the ANOPP system environment. Includes corrective action taken when system conditions are invalid.

Control Statement - (CS) One or more card images which define a particular action to be performed by the ANOPP EM System. A set of control statements defines the execution sequence of an ANOPP run.

Control Statement Processing Phase - The phase of EM execution in which the control statements are processed.

Control Structure - A table, a directory or any other information block which is core resident and not residing on a data unit/member.

CS - see Control Statement

DATA - DATA is the data unit created by ANOPP Executive Management System. It is used to store data members created as a result of the DATA control statement encountered in the Primary Input Stream.

Data Base Management System - (DBM) The subsystem of ANØPP which provides a method of storing and retrieving data on auxiliary storage. Used by the ANOPP executive system and by functional modules.

Data Base Structure - A table, a directory, or any other information block which resides on a data unit. The general organizational structure of a data unit and a data member are also included as data base structures.

Data Element - One or more words residing on a formatted data record. The number of words is determined by the data member format.

Data Member - (DM) An ordered set of information which resides, in a logically continguous fashion, on a data unit. The information includes user data and Data Member Manager data.

Data Member Format - Specification which describes the composition of data records residing on a data member. The specification for a formatted record is a string of element codes.

Data Record - An ordered set of data elements or words residing on a data member. The record may be unformatted or it may be formatted as fixed, variable, or card image according to the data member format.

Data Table - A user-created table of data available to the functional module for processing. A one-record data member having an internal format corresponding to a defined Data Table Type.

Data Table Type 1 - A tabulated function of n independent variables (present maximum = 4) for which acceptable interpolation and extrapolation procedures may be defined.

Data Unit - (DU) A Data Unit is the highest level of the ANOPP Data Base Management System data structure that can be referenced directly by ANOPP modules. It is physically stored on direct access storage devices and is uniquely identified within an ANOPP run by a data unit name. A data unit is a set of data members.

DBM - see Data Base Management System

DM - see Data Member

GLOSSARY

DSM - see Dynamic Storage Management System

DU - see data unit

Dynamic Storage Management System - (DSM) The subsystem of the ANOPP Executive Management System that provides a method of allocating and releasing blocks of core storage within ANOPP.

Element code - Descriptor within a data member format used to describe an element code for a one word integer element.

EM - see Executive Management System

End of Data - (EØD) end of data character ($), recognized by the Executive Cracking Module (XCR) and Executive Crack Without Conversion Module (XCRWC) as the termination of character string data. Utilized primarily as the terminator of control statement images.

EØD - see End of Data

Error Processing Phase - The phase of EM execution which determines the action to be taken when a non-fatal error occurs during the processing of a CS. The action depends on the value of system parameter JCØN. JCØN = .TRUE. results in the resumption of processing with the CS following the CS in error. JCØN = .FALSE. results in the resumption of processing with the next PRØCEED CS or ENDCS CS, whichever occurs first.

Error Termination Phase - The phase of EM execution which results in abnormal termination of ANOPP with an informative message as to the cause. It is entered when an executive module detects an error condition which inhibits further meaningful execution.

Executive Management System - (EM) The subsystem of ANOPP which performs initialization and validation of the ANOPP System, directs the sequence of processing based on a user-supplied CS set, directs action taken after the occurrance of a non-fatal error, and performs a normal or abnormal termination.

F.M. - see Functional Module

Functional Module - (F.M.) One or more executable modules recognized by the ANOPP executive system. A functional module is called into execution when the Control Statement Processing Phase encounters an EXECUTE CS and transfers control to the Function Module Processing Phase.

Functional Module Processing Phase - The phase of EM execution which interrupts the CS Processing Phase and brings into execution the F.M. specified on the EXECUTE CS. Upon completion of the F.M., the integrity of the ANOPP system environment is validated and insured through Cleanup Procedures.

GDS - see Global Dynamic Storage

General Utilities - a collection of general purpose modules available for usage by all executive system routines. Most of the general utility modules are also available for use by functional modules.

Global Dynamic Storage - (GDS) A section of free core storage defined and maintained by DSM to provide for inter-module communication and for storage of ANOPP directories and tables. GDS resides at the end of a user's field length for the life of an ANOPP run. The length is determined by a parameter on the ANØPP CS.

Hierarchy Chart - A graphical representation of the functional relationship between modules.

IDX - An IDX is an integer variable which contains the location of a block of dynamic storage relative to a reference point which, for the ANOPP system, is the /XANØPP/ common block.

GLOSSARY

Index - Location relative to beginning of table or table entry referenced with an ordinal of 1 (one). Used primarily in DBM module prologues and descriptions.

Initialization Phase - The phase of EM execution which controlls the initialization of the ANOPP system environment, which includes printing of the standard ANOPP title page, processing the Primary Input Stream through the STARTCS CS and performing initialization functions for EM, DBM and DSM.

LDS - see Local Dynamic Storage

Local Dynamic Storage - (LDS). That part of core storage maintained by the Dynamic Storage Management System that begins with the word following the longest segment in current execution and ends at the start of GDS.

M001 - The data member name which contains the Primary CS Set on XSUNIT data unit.

Member Manager - (DMM, MM). That part of the DBM sub-system which provides the F.M. writer and the EM subsystem with basic open/close, read/write and position functions for creating, accessing, and maintaining data members.

MM - see Member Manager

Module - A FORTRAN or COMPASS subprogram.

Mxxx - Name of the data member which contains a Secondary or Primary CS Set.

Mxxx Completed Execution - Completed execution describes the status of an Mxxx member when all control statement records on that member have been processed. The Mxxx member is not in current execution or in suspended execution.

Mxxx Current Execution - Describes the status of an Mxxx member when that member is open and the control statement records on the member are being processed. The CS currently in execution is on the Mxxx member.

Mxxx Suspended Execution - An Mxxx member is put into suspended execution if, during processing of the Mxxx member by the CS Processing Phase, a CALL CS is encountered. The Mxxx member in current execution is closed and processing resumes with the Secondary Input Stream specified by the CALL CS. When the Secondary Input Stream is completed, the Mxxx member in suspended execution is reopened and processing resumes with the CS following CALL.

Normal Termination Phase - The phase of ANOPP execution which is entered when CS Processing Phase is complete, and includes printing an informative message, closing member M001, and halting execution.

Parameter Maintenance Functions - A group of general utilities which establish, change, or retrieve user parameter values. These include XPUTP, XASKP, and XGETP.

Position - Word position relative to beginning of table or table entry referenced with an ordinal of 0 (zero). Used primarily in DBM module prologues and descriptions.

Primary CS Set - The executable form of the Primary Input Stream constructed during the Primary Edit Phase. It resides on the root member, M001, on the EM unit XSUNIT.

Primary Edit Phase - The phase of EM execution during which the M001 root member is built from the control statements in the Primary Input Stream. This phase follows the completion of the initialization phase.

Primary Input Stream - The set of card images found in the ANOPP input stream beginning with the first image following the STARTCS control statement and including all images through the ENDCS control statement.

GLOSSARY

Root Member - see Primary CS Set and M001.

Secondary CS Set - The executable form of the Secondary Input Stream constructed during the Secondary Edit Phase and residing on an Mxxx member on XSUNIT.

Secondary Edit Phase - The phase of EM execution during which a CALL control statement is processed. On the first execution of a CALL CS, the Secondary Edit Phase builds an Mxxx type member containing CS records that correspond to control statements in the Secondary Input Stream. The Secondary Edit Phase provides the environment required for the CS Processing Phase to resume execution with the first control statement on the new Mxxx member.

Secondary Input Stream - A set of control statements residing on a data member in card image (CI) format and brought into execution when a CALL CS is processed that specifies the member as the DU (DM) parameter on the CALL CS.

System Parameters - Variables used in determining characteristics of a particular ANOPP run. The default values of certain executive system parameters may be modified during the Initialization Phase via user-supplied values provided on the ANØPP CS. The value of user system parameters may be set during CS Processing Phase via user-supplied values provided on the SETSYS CS.

System Table - A data structure used by various executive modules. The table structure has two parts, a preface and a body. The preface describes the table's current status and the body contains the entries.

Table Manager - (TM). That part of the DBM subsystem which provides open/close, build, and interpolate functions for data tables.

TM - see Table Manager

GLOSSARY

UPDATE Utility - An EM subsystem which provides the ANØPP user with a means of building a new data unit using an existing data unit as a basis for modification or drawing from data members on several data units. It is initiated via the UPDATE CS.

User Parameter - A parameter, when once established, remains available to the user throughout the ANOPP run. The value of a User Parameter may be established or changed during the CS processing phase via the PARAM CS. The value of a user parameter may be established, changed, and retrieved during the F.M. Processing Phase via the Parameter Maintenance Functions.

Utilities - see General Utilities

Uxxx - A member, residing on the EM System scratch unit XSUNIT, containing card image source input data encountered in the Primary Input Stream during the Primary Edit Phase.

XSUNIT - The executive scratch data unit created by the ANOPP Executive System modules. It is reserved for ANOPP Executive System usage.

APPENDIX B

INDEX OF MODULE NAMES

B.1 EXECUTIVE SYSTEM MODULES

The ANOPP executive system is comprised of many modules. Each module is part of the Data Base Management System, the Dynamic Storage Management System, the Executive Management System, the UPDATE Subsystem, or the General Utilities. In the following section, all ANOPP modules are given according to the corresponding executive module in alphabetical order. With the module names are given the Figure number of the hierarchy chart(s) which contains the module, and the section number where a description of the module can be found (if applicable).

INDEX OF MODULE NAMES

B.1.1. Data Base Management System

B.1.1.1. Member Manager

The modules comprising the DBM are listed below. Figure numbers refer to figures in Section 3.6.3

Name	Figure(s)	Section
MMBAME	2,13,14,15	
MMBFSI	3	
MMBFST	3	
MMBFT1	3	
MMBFT8	3	
MMBFT9	3	
MMBMCI	4,13,14,15	
MMBMH	3,14,15	
MMCLØS	1,4	3.6.3.7
MMCLSE	4	
MMCRMX	4,7,11,12,13,16,20,23	
MMDØMC	4,5,13,23	
MMEDNM	1,4,8,9,10,12,17,18,19	
MMERR	6	3.6.3.8.1
MMFEFB	5	
MMGED	8,17,21	
MMGEFB	11	
MMGET	7,8,9,10	
MMGETE	1,8	3.6.3.5.3
MMGETR	1,4,9	3.6.3.5.1
MMGETW	1,10	3.6.3.5.2
MMGNEW	8	
MMGNWE	8	
MMIØMC	11,13,14,15,23	
MMMDMH	4	
MMNWR	12	
MMØPRD	1,4,13	3.6.3.3.1
MMØPWD	1,14	3.6.3.3.2
MMØPWS	1,15	3.6.3.3.3
MMPØSN	1	3.6.3.6.1
MMPUT	16,17,18,19	
MMPUTE	1,17	3.6.3.4.3.

EXECUTIVE SYSTEM MODULES

Name	Figure(s)	Section
MMPUTR	1,4,18	3.6.3.4.1
MMPUTW	1,19	3.6.3.4.2
MMREW	1	3.6.3.6.2
MMRMD	4,13,20,23	
MMRMH	13	
MMRRS	12	
MMSAMD	2	
MMSFEI	8,17,21	
MMSKIP	1	3.6.3.6.3
MMSUD	22,23	
MMUHMD	4	
MMUPMD	4	
MMVBA	4,5,7,12,13,16,20,23	
MMVNM	13,14,15,22	
MMVTD	13,14,15	
MMVUM	24	3.6.3.8.2

INDEX OF MODULE NAMES

B.1.1.2 Table Manager

The modules comprising the TM are listed below. Figure numbers refer to figures in Section 3.6.4.7

Name	Figure(s)	Section
TMBLD1	24	3.6.4.5.1
TMCLØS	24	3.6.4.3
TMEA	26	
TMEAIN	26	
TMEARS	26	
TMEARD	26	
TMEB	26	
TMEBIN	26	
TMEBRS	26	
TMEBRD	26	
TMEDI1	24	
TMEDTB	27	
TMERR	25	3.6.4.6.1
TMFTE	24	
TMGEN1	24	
TMINX	27	
TMINYZ	27	
TMINEX	26,27	
TMLIN	26	
TMLINT	26	
TMLRS	26	
TMLRD	26	
TMMØPN	24	
TMØPN	24	3.6.4.2.2
TMØPNA	24	3.6.4.2.1
TMSRCH	27	
TMSTD	24,27	
TMTERP	24,27	3.6.4.4
TMTABP	27	
TMTBL1	27	
TMTBL2	27	
TMTBL3	27	
TMTØPN	24	
TMVSEQ	24	

EXECUTIVE SYSTEM MODULES

B.1.2 Dynamic Storage Management System

The modules comprising the DSM are listed below. Figure numbers refer to figures in Section 3.7.5

Name	Figure(s)	Section
DSMB	1	3.7.3.1
DSMCAB	3,9	
DSMCØN	4,6,7	
DSMDFB	9	
DSMDLK	3,4,9	3.7.4.1
DSMERR	2	
DSMET	1,2,4,6,7,8	
DSMEUX	2,8,9	
DSMF	1,3,9	3.7.3.2
DSMFLB	4,7	
DSMG	1,4,9	3.7.3.3
DSMGUB	4	
DSMI	1,5	3.7.3.4
DSMIDS	5	
DSML	1,6	3.7.3.5
DSMQ	1,7	3.7.3.6
DSMR	1	3.7.3.7
DSMRDC	4	
DSMRLK	4,9	
DSMRSV	4	
DSMS	1,8,9	3.7.3.8
DSMU	1	3.7.3.9
DSMX	1,9	3.7.3.10
DSMXFB	9	
DSM1ST	3	

INDEX OF MODULE NAMES

B.1.3 Executive Management System

The modules comprising the EM are listed below. Figure numbers refer to figures in Section 3.5

Name	Figure(s)	Section
XAR	6	
XAT	2,6	
XBS	3,14	3.5.4.1
XBSDBM	3	
XBSDFL	3	
XBSDSM	3	
XBSGCS	3	
XBSIN	3	
XBSSP	3	
XBSTP	3	
XCA	4,6	3.5.4.6
XCABST	4	
XCACLØ	4	
XCAI	4	
XCAMST	4	
XCAMXX	4	
XCANCS	4	
XCANS	4	
XCANSP	4	
XCANWC	4	
XCATRA	4	
XCO	6	
XCSCCS	11	
XCSCIL	11	
XCSCRD	11	

EXECUTIVE SYSTEM MODULES

Name	Figure(s)	Section
XCSCRS	11	
XCSIL	4,6	
XCSINT	16	
XCSLØG	6	
XCSP	6,14	3.5.4.3
XCSPM	4,6	
XCSRD	16	
XCSRS	16	
XCSSL	6,11	
XCSST	11,16	
XCT	6,7	
XCTBDU	2,3,7	
XCTBMD	3,7	
XCTDU	3,7	
XCTEFN	2,3,7	
XDR	6,8	
XDT	6	
XEN	6	3.5.4.7
XEX	6,9	
XEXA	9	
XEXL	9	
XFM	1,10	3.5.4.4
XFMANT	10	
XFMDSM	10	
XFMMM	10	
XFMTM	10	
XGØ	6	
XIF	6,11	
XLD	6,12	

INDEX OF MODULE NAMES

EXECUTIVE SYSTEM MODULES

Name	Figure(s)	Section
XRTCSS	17,18	
XRTDAT	18	
XRTEND	18	
XRTI	17	
XRTLRF	5,17	
XRTLSA	4,17	
XRTLSE	4,5,17	
XRTØDB	5,17,18,19	
XRTPIN	18	
XRTRS	17	
XRTSAR	19	
XRTSAT	19	
XRTSCA	19	
XRTSCØ	19	
XRTSCR	19	
XRTSDA	19	
XRTSDR	19	
XRTSDT	19	
XRTSEN	19	
XRTSER	19	
XRTSEX	19	
XRTSGØ	19	
XRTSIF	19	
XRTSLD	19	
XRTSPA	19	
XRTSPR	19	
XRTSPU	19	
XRTSRE	19	
XRTSSS	19	

INDEX OF MODULE NAMES

B.1.4. UPDATE EM Subsystem

The modules comprising the EM are listed. Figure numbers refer to figures in Section 3.8.8

Name	Figure(s)	Section
XUP	6,1	
XUPADD	1	
XUPADS	5	
XUPALL	1	
XUPCDT	2	
XUPCGP	2	
XUPCHG	1,2	
XUPCHI	5	
XUPCHS	5	
XUPCHX	5	
XUPCIN	2	
XUPCØB	1,2	
XUPCØS	5	
XUPCPY	1	
XUPCQD	5	
XUPCQT	2	
XUPCS	1	
XUPDIR	5	
XUPECE	5	
XUPECI	4	
XUPERR	3	3.8.7.1
XUPGPR	2,6	
XUPINS	4,5	
XUPLST	1	
XUPMLV	4,5	
XUPNEW	1	

INDEX OF MODULE NAMES

B.1.5 General Utilities

The modules comprising the General Utilities are listed below. Figure numbers refer to figures in Section 3.9.4

Name	Figure(s)	Section
ALPHA	1	3.9.2.1
DIGIT	1	3.9.2.2
DVALUE	1	3.9.2.3
IAND	1	3.9.2.4
ICD	1	3.9.2.5
ICI	1	3.9.2.6
ICØMPL	1	3.9.2.7
IDATE	1	3.9.2.8
ILØC	1	3.9.2.9
ILSHFT	1	3.9.2.10
IMASK	1	3.9.2.11
IØR	1	3.9.2.12
IRSHFT	1	3.9.2.13
ISHIFT	1	3.9.2.14
ITIME	1	3.9.2.15
IVALUE	1	3.9.2.16
IXØR	1	3.9.2.17
MEMNUM	1	3.9.2.18
NUMTYP	1	3.9.2.19
NWDTYP	1	3.9.2.20
RVALUE	1	3.9.2.21
XASKP	1	3.9.2.22
XBSRIN	1	3.9.3.23
XBSRRD	1	3.9.2.24
XBSRRS	1	3.9.2.25

INDEX OF MODULE NAMES

EXECUTIVE SYSTEM MODULES

Name	Figure(s)	Section
XINC	1	3.9.2.34
XMØVE	1	3.9.2.34
XMPRT	1	3.9.2.35
XPAGE	1	3.9.2.36
XPK	1	3.9.2.37
XPKM	1	3.9.2.38
XPLAB	1	3.9.2.39
XPLABQ	1	3.9.2.40
XPLINE	1	3.9.2.41
XPUTP	1	3.9.2.42
XSORTF	1	3.9.2.43
XSTORE	1	3.9.2.44
XTBDMP	1	3.9.2.45
XTBERR	10	3.9.3.2
XTRACE	1, 11	3.9.2.46
XTRLØC	11	
XT1AL	1	3.9.2.47
XT1FV	1	3.9.2.48
XT2AL	1	3.9.2.49
XT3FL	1	3.9.2.50
XT3FV	1	3.9.2.51
XT3IF	1	3.9.2.52
XT3LK	1	3.9.2.53
XUFMSG	12	3.9.3.1
XUNPK	1	3.9.2.54
XUNPKM	1	3.9.2.55
XUNPKT	1	3.9.2.56
XVNAME	1	3.9.2.57
XZFILL	1	3.9.2.58

APPENDIX C

INDEX TO ERROR MESSAGE NUMBERS

C.1 EXECUTIVE SYSTEM ERRORS

Following is a list of the error messages and numbers for the Data Base Management System, Dynamic Storage Management System, Executive Management System, General Utilities, and Update.

The lower case letter n has been used where a FORTRAN name would be printed. The lower case letter v has been used where a FORTRAN value would be printed.

C.1.1 Executive Management System (EM)

C.1.1.1 Fatal Errors

Fatal EM errors are processed by XXFMSG. For further description of the XXFMSG module, see Section 3.5.5.1. Fatal EM errors have been assigned the numbers 1-999.

All messages are prefixed by:

*** EXEC ERROR (ERROR NUMBER v) *** (CALLER n)

The messages and numbers are as follows:

1 INSUFFICIENT CORE TO EXPAND TABLE. CURRENT LENGTH OF n TABLE IS v.

2 ERROR IN ANOPP PRIMARY EDIT PHASE.

3 INSUFFICIENT CORE TO ALLOCATE n TABLES.

4 MEMCUR, n, IS NOT THE SAME AS THE NAME, n, IN THE MDBT.

5 INVALID ODB ENTRY. TYPE CODE = v.

6 INVALID MEMBER TYPE OR MAX NUMBER OF MEMBERS EXCEEDED. TYPE = n.

7 KRACKED TABLE OVERFLOW. RECOMPILATION NECESSARY TO ALLOW FOR v CARD IMAGES PER CONTROL STATEMENT.

8 MAXIMUM RECORD LENGTH, v, RETURNED FROM MEMBER MANAGER OPEN CALL FOR MEMBER n IS NOT THE SAME AS THE MAX CS RECORD LENGTH, v, IN THE MDBT OR THE LABEL RECORD LENGTH, v, IN THE MDBT.

9 STATUS, v, RETURNED FROM MEMBER MANAGER POSITION CALL FOR MEMBER n.

INDEX TO ERROR MESSAGE NUMBERS

10 FOR MEMBER n STATUS RETURNED FROM MEMBER MANAGER GET RECORD CALL IS v NUMBER OF WORDS EXPECTED v -- NUMBER OF WORDS RETURNED v.

11 REQUESTED LABEL n IS NOT FOUND.

12 INVALID CONTROL STATEMENT NAME, n, ON CURRENT MEMBER n.

13 INVALID INPUT n = v.

14 INVALID INPUT n = n.

15 UNEXPECTED ERROR RETURNED FROM MEMBER MANAGER CALL. CODE IS v.

16 INVALID INTEGER, v, USED IN IDENTIFICATION OF EXECUTIVE SYSTEM MODULE OR FUNCTIONAL MODULE.

17 ERROR DETECTED IN ANOPP INITIALIZATION PHASE

18 UNEXPECTED OUTPUT FROM n. PARAMETER IS n - VALUE IS v.

19 UNEXPECTED OUTPUT FROM n. PARAMETER IS n - VALUE IS n.

20 END OF FILE DETECTED IN PRIMARY INPUT STREAM. INSUFFICIENT INPUT FOR REQUIRED STARTCS.

21 THE LRCS BLOCK ALLOCATED IS NOT MAXIMUM REQUIRED FOR XCA PROCESSING.

22 NONEXPANDABLE TABLE n IS INSUFFICIENT.

23 SUBSTITUTION TABLE ALLOCATION IS NOT EXACT NUMBER OF WORDS MOVED TO TABLE.

C.1.1.2 Non-Fatal Errors

Non-fatal EM errors are processed by XXNMSG. For further description of the XXNMSG module, see Section 3.5.5.2. VAR4 is a ten word array which enables the printing of card images and more explanatory error messages. Non-fatal EM errors have been assigned the numbers 1001-1999.

All messages are prefixed by:

*** EXEC ERROR (ERROR NUMBER v) *** (CALLER n)

The messages and numbers are as follows:

1001 INVALID LABEL FIELD.

1002 INVALID OR MISSING CS NAME

1003 CONTINUATION SEQUENCE EXCEEDS MAXIMUM CARD LIMIT. SEQUENCE IS ARBITRARILY TERMINATED.

1004 REFERENCE MADE TO NON-EXISTENT LABEL = n.

1005 DUPLICATE n = n.

1006 GOTO OR IF CS REFERENCES OWN LABEL = n.

1007 INVALID END* FIELD OR EXTRANEOUS FIELDS DETECTED ON END* CS.

1008 EXTRANEOUS FIELDS DETECTED ON n CONTROL STATEMENT.

1009 EOF DETECTED ON INCOMPLETE CS IN INPUT STREAM. IT IS CONSIDERED IN ERROR AND IS NOT BEING PROCESSED.

1010 INVALID CS NAME = n.

1011 STARTCS CONTROL STATEMENT MISSING. COMPILATION CONTINUING.

1012 INVALID STARTCS CONTROL STATEMENT. COMPLETE ERROR RECOVERY.

1013 KEYWORD FIELD IS NOT A NAME OR MISSING = SIGN. PROCESSING CONTINUES WITH NEXT ENCOUNTERED VALID FIELD.

1014 THE INITIALIZATION VALUE FOR n IS INCORRECT. PROCESSING CONTINUES WITH NEXT ENCOUNTERED VALID FIELD.

1015 THE LENGTH OF GLOBAL CORE REQUESTED FOR THIS ANOPP RUN IS v. THE MINIMUM LENGTH REQUIRED IS v.

1016 n IS AN INVALID KEYWORD. PROCESSING CONTINUES WITH NEXT ENCOUNTERED VALID FIELD.

1017 NEXT TO LAST FIELD ON ANOPP CS IS EXTRANEOUS.

INDEX TO ERROR MESSAGE NUMBERS

1018 LAST FIELD ON ANOPP CS IS EXTRANEOUS.

1019 INSUFFICIENT CORE TO EXPAND TABLE. CURRENT LENGTH OF n IS v.

1020 USER PARAMETER n NOT FOUND IN USER PARAMETER TABLE.

1021 ATTEMPT TO PERFORM n OPERATION ON TWO FIELDS OF DIFFERENT TYPES. TYPES ARE v AND v.

1022 ILLEGAL FIELD TYPE FOR n OPERATION. TYPE IS v.

1023 ATTEMPT TO COMPARE TWO n FIELDS WITH INVALID LOGICAL OPERATOR. CODE FOR OPERATOR IS v.

1024 EXECUTIVE ERROR INDICATOR SET TO .TRUE. WHEN PROCESSING n CONTROL STATEMENT.

1025 MISSING FIELD DETECTED ON n CONTROL STATEMENT.

1026 INVALID FIELD DETECTED ON n CONTROL STATEMENT. FIELD EXPECTED TO CONTAIN n.

1027 UNRECOGNIZABLE FIELD DETECTED n.

1028 UNEXPECTED INPUT. PARAMETER IS n - VALUE IS v.

1029 STARTCS ENCOUNTERED ON ANOPP CONTROL STATEMENT. COMPLETE ERROR RECOVERY.

1030 DUPLICATE MEMBERS DETECTED ON ABOVE DATA CONTROL STATEMENT. MEMBER NAME = n.

1031 TABLE EXPANSION ON n TABLE UNSUCCESSFUL. n ENTRY NOT ADDED.

1032 UPDATE OR TABLE (SOURCE = *) CS FORM IS INVALID IN SECONDARY INPUT STREAM.

1033 SECONDARY INPUT STREAM MEMBER, n, DOES NOT EXIST.

1034 SECONDARY INPUT STREAM MEMBER, n, IS NOT IN CARD IMAGE (CI) FORMAT.

1035 LOCAL DYNAMIC STORAGE INSUFFICIENT TO ALLOCATE ALL BLOCKS NECESSARY FOR SECONDARY INPUT STREAM PROCESSING.

1036 LOCAL DYNAMIC STORAGE HAS BEEN INITIALIZED BUT NOT RELEASED.

1037 USER LOCK ON n HAS NOT BEEN CLEARED.

1038 DATA TABLE n (n) OPENED BUT NOT CLOSED, TABLE REMOVED FROM CORE.

1039 DATA MEMBER n (n) OPENED BUT NOT CLOSED. LOGICAL CLOSE PERFORMED.

1040 DATA UNIT n, DATA MEMBER n CANNOT BE OPENED TO READ.

C.1.2 Data Base Management System (DBM)

C.1.2.1 Member Manager (MM)

Member Manager module and control statement error messages are processed by MMERR. For further description of the MMERR module, see Section 3.6.3.9.1.

All messages are prefixed by:

```
*** DBM ERROR )ERROR NUMBER v) *** (CALLER n)
```

The messages and numbers are as follows:

1 BAD INPUT TO SUBROUTINE n. VALUE = v.

2 DATA UNIT NAME NOT UNIQUE.

3 EXTERNAL FILE NAME NOT UNIQUE.

4 DATA UNIT DIRECTORY FULL.

5 DATA MEMBER DIRECTORY SPACE NOT AVAILABLE. LENGTH OF MEMBER DIRECTORY IS n. LENGTH OF DYNAMIC STORAGE BLOCK OBTAINED FOR THE MEMBER DIRECTORY IS v.

6 BUFFER SPACE NOT AVAILABLE.

7 BUFFER ALREADY EXISTS.

8 BUFFER DOES NOT EXIST.

9 FILE DOES NOT HAVE PROPER HEADER.

10 DATA UNIT DOES NOT EXIST.

11 CANNOT GENERATE EXTERNAL FILE NAME.

12 DATA UNIT DIRECTORY SPACE IS NOT AVAILABLE.

13 DATA TABLE DIRECTORY SPACE IS NOT AVAILABLE.

14 ACTIVE MEMBER DIRECTORY SPACE IS NOT AVAILABLE.

15 MEMBER DIRECTORY SPACE IS NOT AVAILABLE.

16 XSUNIT NOT CREATED.

17 DATA NOT CREATED.

18 OPEN MEMBER COUNT NOT ZERO.

19 LAST RECORD NOT COMPLETE ON UNIT n MEMBER n.

INDEX TO ERROR MESSAGE NUMBERS

20 LIBRARY FILE DIRECTORY SPACE NOT AVAILABLE.

21 DATA MEMBER IS ALREADY OPEN ON UNIT n MEMBER n.

22 DATA UNIT IS NOT IN THE UNIT DIRECTORY. UNIT n MEMBER n.

23 ACTIVE MEMBER DIRECTORY IS FULL AND CANNOT BE EXPANDED - EXPAND FIELD LENGTH AND GLOBAL DYNAMIC STORAGE.

24 DATA UNIT ALREADY OPEN FOR DIRECT WRITE. UNIT n MEMBER n.

25 WRITE INHIBITED ON DATA UNIT. UNIT n MEMBER n.

26 INSUFFICIENT DYNAMIC STORAGE FOR MEMBER MANAGER USE - EXPAND FIELD LENGTH AND GLOBAL DYNAMIC STORAGE. UNIT n MEMBER n.

27 THE DATA MEMBER IS OPEN TO TABLE MANAGER. UNIT n MEMBER n.

28 DATA MEMBER IS NOT IN MEMBER DIRECTORY. UNIT n MEMBER n.

29 OPEN MEMBER COUNT IS NEGATIVE. UNIT n.

30 DATA UNIT OR MEMBER NAME IS MALFORMED. UNIT n.

31 READ ERROR ON MEMBER DIRECTORY. UNIT n.

32 READ ERROR ON MEMBER HEADER. UNIT n MEMBER n.

33 INVALID DIRECTORY OR TABLE ID.

34 INVALID MODE ARGUMENT INPUT. UNIT n MEMBER n.

35 INVALID UNIT DIRECTORY ENTRY. UNIT n.

36 FILE BUFFER ASSIGNMENT FAILED. UNIT n.

37 INVALID INPUT ARGUMENT. UNIT n MEMBER n.

38 INVALID MEMBER FORMAT. UNIT n MEMBER n. FORMAT SPECIFICATION IMAGE FOLLOWS. n ... n.

39 MISMATCHED RIGHT AND LEFT PARENTHESES. UNIT n MEMBER n.

40 UNRECOGNIZABLE FIELD(S) IN FORMAT. UNIT n MEMBER n.

41 FORMAT FIELD - TYPE = n, VALUE = v.

42 INVALID VARIABLE FORMAT - VARIABLE REPEAT GROUP MUST BE LAST. UNIT n MEMBER n.

43 INVALID REPEAT GROUP SPECIFICATION - FORMAT IS INVALID. UNIT n MEMBER n.

44 INVALID ELEMENT SPECIFICATION - FORMAT IS INVALID. UNIT n MEMBER n.

45 EXTRANEOUS LEFT PARENTHESIS. UNIT n MEMBER n.

46 UNIDENTIFIABLE SEPARATOR IN FORMAT. UNIT n MEMBER n.

EXECUTIVE SYSTEM ERRORS

47 MORE THAN ONE VARIABLE REPEATING GROUP SPECIFIED. UNIT n MEMBER n.

48 CYBER RECORD MANAGER ERROR ON PUT. UNIT n MEMBER n.

49 PREVIOUS RECORD IS INCOMPLETE. UNIT n MEMBER n.

50 RECORD LENGTH IS INCOMPATIBLE WITH FORMAT SPECIFICATION. UNIT n MEMBER n.

51 NUMBER OF RECORDS PUT TO MEMBER EXCEEDS MAXIMUM DEFINED BY THE OPEN REQUEST. UNIT n MEMBER n.

52 NUMBER OF WORDS TO BE PUT IS NEGATIVE. UNIT n MEMBER n.

53 UNUSED.

54 UNUSED.

55 LAST I/O OPERATION DID NOT END ON AN ELEMENT BOUNDARY. UNIT n MEMBER n.

56 THIS CALL MAY NOT BE USED WITH UNFORMATTED RECORDS. UNIT n MEMBER n.

57 NUMBER OF ELEMENTS TO BE PUT IS NEGATIVE. UNIT n MEMBER n.

58 TOTAL RECORD LENGTH EXCEEDS FIXED FORMAT SPECIFICATION. UNIT n MEMBER n.

59 RECORD DIRECTORY IS FULL - INCREASE MAXIMUM NUMBER OF RECORDS IN OPEN REQUEST. UNIT n MEMBER n.

60 NUMBER OF WORDS TO BE READ IS LESS THAN OR EQUAL TO ZERO. UNIT n MEMBER n.

61 ATTEMPT TO READ BEYOND END OF MEMBER. UNIT n MEMBER n.

62 CYBER RECORD MANAGER ERROR ON GET. UNIT n MEMBER n.

63 RECORD ARRAY SIZE IS LESS THAN OR EQUAL TO ZERO. UNIT n MEMBER n.

64 NUMBER OF ELEMENTS TO BE READ IS LESS THAN OR EQUAL TO ZERO. UNIT n MEMBER n.

65 MEMBER IS UNFORMATTED - IMPROPER USE OF THIS CALL. UNIT n MEMBER n.

66 NUMBER OF WORDS READ IS INCOMPATIBLE WITH THE FORMAT ELEMENT SPECIFICATIONS. UNIT n MEMBER n.

67 NAME(3) IS NOT A VALID IDX. UNIT n MEMBER n.

68 INVALID DATA MEMBER NAME OR IDX IN NAME ARGUMENT. UNIT n MEMBER n.

69 INVALID DATA UNIT NAME OR IDX IN NAME ARGUMENT. UNIT n MEMBER n.

70 DATA MEMBER IS NOT OPEN FOR THE MODE SPECIFIED FOR THIS CALL. UNIT n MEMBER n.

71 WARNING - OLD MEMBER IS STILL OPEN TO READ FOLLOWING CLOSE OF NEW MEMBER OF SAME NAME. UNIT n MEMBER n.

72 ATTEMPTED TO CLOSE MEMBER OPEN VIA MMOPWS WHILE ANOTHER MEMBER ON THE SAME UNIT WAS STILL OPEN VIA MMOPWD. UNIT n MEMBER n.

73 INSUFFICIENT GLOBAL DYNAMIC STORAGE FOR MMCLOS SCRATCH COPY. UNIT n MEMBER n.

74 INVALID NAME ARGUMENT INPUT. UNIT n MEMBER n.

75 INVALID MODE ARGUMENT INPUT. UNIT n MEMBER n.

The LØAD control statement uses the module XLDERR to process its errors.

The messages processed by XLDERR are listed below. For further description of the XLDERR module, see Section 3.6.3.8.3

All messages are prefixed by:

*** LOAD ERROR (ERROR NUMBER - v) *** (CALLER - n)

The messages and numbers are as follows:

1 ERROR v DETECTED BY OPERATING SYSTEM ON FILE n.

2 INSUFFICIENT LOCAL DYNAMIC CORE IS AVAILABLE.

3 EXTERNAL FILE NAME n IS ALREADY ASSIGNED TO DATA UNIT n.

4 THE LIBRARY DIRECTORY RECORD IS INVALID. ID IS n, CNE IS v.

5 DATA UNIT n IS NOT DEFINED ON SEQUENTIAL LIBRARY n.

6 DATA UNIT n, DATA MEMBER n IS NOT DEFINED ON SEQUENTIAL LIBRARY n.

7 NUMBER OF DATA UNITS TO BE LOADED EXCEEDS THE NUMBER OF ENTRIES AVAILABLE IN THE DATA UNIT DIRECTORY.

8 DATA UNIT n ALREADY EXISTS IN THE DATA UNIT DIRECTORY.

9 FILE NAME /n/ GIVEN FOR DATA UNIT n IS ALREADY IN USE FOR ANOTHER DATA UNIT.

10 FILE NAME /n/ GIVEN FOR DATA UNIT n IS IN USE AS A LIBRARY FILE.

11 LIBRARY FILE ANOMALY, DATA UNIT/MEMBER NAMED IN THE LIBRARY DIRECTORY IS NOT ON THE LIBRARY.

12 LIBRARY UNIT HEADER IS INVALID v.

The UNLØAD control statement uses the module XUNERR to process its errors.

The messages processed by XUNERR are listed below. For further description of the XUNERR module, see Section 3.6.3.8.4

All messages are prefixed by:

EXECUTIVE SYSTEM ERRORS

*** UNLOAD ERROR (ERROR NUMBER - v)   *** (CALLER - n)

The messages and numbers are as follows:

1 ERROR v DETECTED BY OPERATING SYSTEM ON FILE n.

2 INSUFFICIENT LOCAL DYNAMIC CORE IS AVAILABLE.

3 EXTERNAL FILE NAME n IS ALREADY ASSIGNED TO DATA UNIT n.

4 SEQUENTIAL LIBRARY FILE n WAS USED IN PREVIOUS LOAD OR UNLOAD.

5 DATA UNIT n IS NOT DEFINED.

6 DATA MEMBER n DOES NOT EXIST ON DATA UNIT n.

The DROP control statement prints its own message for errors encountered.

INDEX TO ERROR MESSAGE NUMBERS

C.1.2.2 Table Manager (TM)

Table Manager and TABLE control statement error messages are processed by TMERR. For further description of the TMERR module, see Section 3.6.4.6.1.

All messages are prefixed by;

*** DTM ERROR (ERROR NUMBER v) *** (CALLER n)

The messages and numbers are as follows:

1 ARGUMENT v OUT OF RANGE. ARGUMENT IS n.

2 INVALID INPUT

3 DYNAMIC CORE NOT AVAILABLE.

4 ERROR RETURNED FROM n. ERROR CODE IS v.

5 INDEPENDENT VARIABLE ARRAY NOT IN MONOTONIC SEQUENCE. ARRAY DUMP FOLLOWS v.

6 INPUT RECORD FROM UNIT n, MEMBER n, NOT IN CARD IMAGE FORMAT.

7 LOCAL CORE BLOCK FOR CRACK TABLE NOT SUFFICIENT.

8 VALUE n NEEDED TO BUILD TABLE TYPE n NOT PRESENT.

9 INVALID VARIABLE NAME n.

10 INVALID INPUT - CURRENT CARD IMAGE IS NOT CARD EXPECTED.

11 INVALID INPUT - DUPLICATE VARIABLE NAME n.

12 ERROR DETECTED ON FOLLOWING CARD IMAGE v.

13 INVALID INPUT - MISSING FIELDS DETECTED.

14 INVALID INPUT - FIELD EXPECTED n.

15 TABLE NOT BUILT.

16 INVALID VALUE ENTRY, EXPECTED VARIABLE TYPE n - ACTUAL VARIABLE TYPE v.

17 EXCESSIVE VALUE ENTRIES.

18 TABLE ON UNIT - n AND MEMBER - n IS NOT OPEN AS EXPECTED.

19 USER DOESN'T OWN DATA TABLE.

20 IDX TO THE DATA TABLE DIRECTORY IS INVALID.

21 NAMED DATA TABLE, n; n, IS ALREADY OPEN TO DATA TABLE MANAGER.

EXECUTIVE SYSTEM ERRORS

22 NAMED DATA TABLE, n, n, IS ALREADY OPEN TO DATA MEMBER MANAGER.

23 THE DATA TABLE DIRECTORY IS FULL.

24 THE NAMED DATA TABLE n, n IS NOT DEFINED TO DATA BASE MANAGER.

25 SUFFICIENT DYNAMIC STORAGE IS NOT AVAILABLE FOR DATA TABLE n, n.

26 DATA MEMBER n, n, IS NOT A DTM DATA TABLE.

27 DATA TABLE n, n IS NOT DEFINED TO DATA TABLE MANAGER.

28 NAME ARGUMENT USED TO CLOSE DATA TABLE n, n MUST BE THE ONE USED IN OPENING THE TABLE.

C.1.3 Dynamic Storage Management System (DSM)

DSM error messages are processed by DSMERR. For further description of the DSMERR module, see Section 3.7.4.1.

All messages are prefixed by:

*** DSM ERROR (ERROR NUMBER v) *** (CALLER n)

The messages and numbers are as follows:

1 n IS ALREADY INITIALIZED.

2 n CORE IS INSUFFICIENT FOR INITIALIZATION.

3 GDS/LDS OVERLAP.

4 n IS INVALID DS TYPE.

5 v IS INVALID DS START ADDRESS.

6 n IS INVALID DS USER.

7 MIN IS GREATER THAN MAX.

8 INVALID IDX FOR n.

9 n IS ALREADY UNLOCKED.

10 n HAS ALREADY BEEN OVERLAYED.

11 n IS NOT INITIALIZED.

12 MIN OR MAX IS INVALID NEGATIVE LENGTH.

13 MIN AND MAX ARE ZERO LENGTH.

C.1.4 UPDATE

UPDATE error messages are processed by XUPERR. For further description of the XUPERR module, see Section 3.8.7.1.

All messages are prefixed by:

*** UPDATE ERROR (ERROR NUMBER v) *** (CALLER n)

The messages and numbers are as follows:

1 INVALID INPUT = v.

2 MEMBER MANAGER OPEN TO READ FOR DATA UNIT n, DATA MEMBER n.

3 INVALID MEMBER LEVEL DIRECTIVE n.

4 INSUFFICIENT CORE TO ALLOCATE v WORDS.

5 UNIT = n IS NOT IN UNIT DIRECTORY.

6 INVALID KEYWORD FIELD.

7 THE ALL PARAMETER MAY NOT BE SPECIFIED DURING CREATE MODE.

8 OLDU, NEWU, OR SOURCE UNITS ARE IN CONFLICT.

9 REQUIRED DATA UNIT IS NOT SPECIFIED.

10 CURRENT CONTROL STATEMENT NAME IS n.

11 INVALID KEYWORD = n.

12 DATA UNIT n IS ARCHIVED.

13 INVALID LIST OPTION = n.

14 NUMBER OF RECORDS OR LENGTH OF ARRAY TO HOLD RECORDS IN TRANSIT IS INCORRECT.

15 SOURCE DATA MEMBER CONTAINING THE SET OF UPDATE DIRECTIVES IS NOT IN CARD IMAGE FORMAT.

16 INCORRECT TABLE NAME FOR n TABLE.

17 DATA UNIT n, DATA MEMBER n IS NOT OPEN TO READ.

18 DATA UNIT n, DATA MEMBER n IS NOT OPEN TO WRITE.

19 NUMBER OF RECORDS TO BE COPIED v OR LENGTH OF ARRAY v IS INCORRECT.

20 DATA MEMBER n ALREADY EXISTS ON THE NEW DATA UNIT n.

INDEX TO ERROR MESSAGE NUMBERS

21 NUMBER OF RECORDS v COPIED TO THE NEW DATA MEMBER IS LESS THAN NUMBER OF RECORDS v REQUESTED.

22 BAD FIELDS RETURNED FROM CALL TO XCR.

23 INVALID RECORD READ FROM DATA MEMBER UPDATS.

24 n IS AN INVALID RECORD LEVEL DIRECTIVE.

25 NUMBER OF CARD IMAGES EXCEEDS MAXIMUM ALLOWED v.

26 DIRECTIVE INCOMPLETE WHEN END OF SOURCE MEMBER ENCOUNTERED. COMPLETE ERROR RECOVERY.

27 INSUFFICIENT STORAGE FOR CRACKING UPDATE DIRECTIVE.

28 INSUFFICIENT CORE FOR STORING THE UPDATE DIRECTIVE IMAGE.

29 DIRECTIVE CONTAINS A FIELD OF IMPROPER TYPE. FIELD SHOULD CONTAIN A NAME.

30 DATA UNIT n, DATA MEMBER n SPECIFIED TO BE COPIED DOES NOT EXIST ON OLD UNIT.

31 UPDATE DIRECTIVE CONTAINS A FIELD OF IMPROPER TYPE. FIELD SHOULD CONTAIN AN INTEGER.

32 EXTRANEOUS FIELDS ON UPDATE DIRECTIVE.

33 LAST RECORD TO BE COPIED v IS NOT IN THE RANGE OF RECORDS ON OLDM.

34 INVALID FORM = KEYWORD FIELD.

35 RECORD v TO BE INSERTED IS NOT WITHIN THE RANGE OF RECORDS ON OLDM.

36 INSUFFICIENT CORE TO ALLOCATE OR EXPAND n TABLE.

37 THE OLDM KEYWORD FIELD IS NOT FOLLOWED BY A DATA UNIT, A DATA MEMBER, OR *.

38 INVALID NEWM KEYWORD FIELD. NEWM = MUST BE FOLLOWED BY A MEMBER NAME.

39 INVALID FORMAT FIELD.

40 INVALID MNR KEYWORD FIELD. MNR = MUST BE FOLLOWED BY AN INTEGER.

41 REQUIRED KEYWORD OLDM IS NOT PRESENT.

42 REQUIRED KEYWORD NEWM IS NOT PRESENT.

43 OLDM = n (IS NOT FOLLOWED BY A DATA MEMBER NAME).

44 DATA MEMBER NAME n IS NOT FOLLOWED BY CLOSING PARENTHESIS

45 THE OLDM KEYWORD FIELD IS NOT FOLLOWED BY A DATA UNIT OF A MEMBER NAME.

46 THE -DELETE RECORD LEVEL DIRECTIVE DOES NOT INCLUDE RECORD NUMBERS.

47 LAST RECORD v TO BE DELETED IS NOT GREATER THAN THE FIRST RECORD v.

EXECUTIVE SYSTEM ERRORS

48 RECORD v AFTER WHICH RECORDS ARE TO BE INSERTED, IS LT OLDM REFERENCE POINTER v.

49 INVALID INPUT n = v.

50 RANGE v = v OF RECORDS TO BE INSERTED NOT WITHIN RANGE OF OLDM.

51 SOURCE DATA UNIT n, DATA MEMBER n CANNOT BE FOUND.

52 OLDM DATA UNIT n, DATA MEMBER n CANNOT BE FOUND.

C.1.5 General Utilities

Most General Utility fatal error messages are processed by XUFMSG. However, several utility modules which are called mainly by Table Manager in maintaining data tables utilize a separate error message module, XTBERR.

The messages processed by XUFMSG are listed below. For futher description of the XUFMSG module, see Section 3.9.3.1.

All messages are prefixed by:

*** UTILITY ERROR (ERROR NUMBER v) *** (CALLER n)

The messages and numbers are as follows:

1 ARGUMENT n OUT OF RANGE. ARGUMENT IS v.

2 ARGUMENT n IS NOT A NUMERIC CHARACTER AS EXPECTED. ARGUMENT IS v.

3 INVALID INPUT n = v.

4 ARGUMENT n IS AN INVALID ANOPP TYPE CODE. ARGUMENT IS v.

5 USER PARAMETER n NOT FOUND IN USER PARAMETER TABLE.

6 USER PARAMETER TYPE v DOESN'T MATCH TYPE IN UPT v.

7 REQUEST TO EXPAND n NOT COMPLETE.

8 INVALID INPUT n = n.

9 n DYNAMIC CORE NOT AVAILABLE FOR n TABLE ALLOCATION.

For further description of the XTBERR module, see Section 3.9.3.2.

All messages are prefixed by:

*** XTB ERROR (ERROR NUMBER v) *** (CALLER n)

The messages and numbers are as follows:

1 INVALID CHAIN IDENTIFIER - n = v.

2 INVALID KEY POSITION - n = v.

3 INVALID POSITION INDICATOR - n = v.

4 NO ROOM FOR NEW ENTRIES - n = v.

5 INVALID SYSTEM TABLE TYPE - n = v.

APPENDIX D

REFERENCES

Publications referenced in this manual are listed below in alphabetic order.

CYBER Record Manager Reference Manual

Control Data Corporation

Publication: 60307300

CYBER Record Manager User's Guide

Control Data Corporation

Publication: 60359600

FORTRAN Extended Version 4 Reference Manual

Control Data Corporation

Publication: 60305601

NASTRAN PROGRAMMERS MANUAL

National Aeronautic and Space Administration

Publication: SP-223 (01)

NOS 1.0 Reference Manual

Control Data Corporation

Publication: 60435400

Update Reference Manual

Control Data Corporation

Publication: 60342500